Le Programmeur

L'art du développement Android

Mark Murphy

2e édition

Édition française établie avec le concours de eXpertise Android

PEARSON

LE PROGRAMMEUR

L'art du développement Android 2

2e édition

Mark L. Murphy

Traduit par Éric Jacoboni
avec la contribution d'Arnaud Farine

PEARSON

Pearson Education France a apporté le plus grand soin à la réalisation de ce livre afin de vous fournir une information complète et fiable. Cependant, Pearson Education France n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Les exemples ou les programmes présents dans cet ouvrage sont fournis pour illustrer les descriptions théoriques. Ils ne sont en aucun cas destinés à une utilisation commerciale ou professionnelle.

Pearson Education France ne pourra en aucun cas être tenu pour responsable des préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces exemples ou programmes.

Tous les noms de produits ou marques cités dans ce livre sont des marques déposées par leurs propriétaires respectifs.

Le logo reproduit en page de couverture et sur les ouvertures de chapitres a été créé par Irina Blok pour le compte de Google sous la licence Creative Commons 3.0 Attribution License **http://creativecommons.org/licences/by/3.0/deed.fr**.

Publié par Pearson Education France
47 bis, rue des Vinaigriers
75010 PARIS
Tél. : 01 72 74 90 00
www.pearson.fr

Mise en pages : TyPAO

ISBN : 978-2-7440-4159-4
Copyright © 2010 Pearson Education France
Tous droits réservés

Titre original : *Beginning Android 2*

Traduit par Éric Jacoboni, avec la contribution technique d'Arnaud Farine

ISBN original : 978-1-4302-2629-1
Copyright © 2010 by Mark L. Murphy
All rights reserved

Édition originale publiée par
Apress
2855 Telegraph Avenue, Suite 600
Berkeley, CA 94705 USA

Aucune représentation ou reproduction, même partielle, autre que celles prévues à l'article L. 122-5 2° et 3° a) du code de la propriété intellectuelle ne peut être faite sans l'autorisation expresse de Pearson Education France ou, le cas échéant, sans le respect des modalités prévues à l'article L. 122-10 dudit code.

Sommaire

À propos de l'auteur XVII

Remerciements XIII

Préface à l'édition française XV

Introduction 1

1. Tour d'horizon 3

2. Projets et cibles 7

3. Création d'un squelette d'application 19

4. Utilisation des layouts XML 27

5. Utilisation des widgets de base 33

6. Conteneurs 45

7. Widgets de sélection 65

8. S'amuser avec les listes 83

9. Utiliser de jolis widgets et de beaux conteneurs 109

10. Le framework des méthodes de saisie 135

11. Utilisation des menus 145

12. Polices de caractères 157

13. Intégrer le navigateur de WebKit 163

14. Affichage de messages surgissants 171

15. Utilisation des threads 177

16. Gestion des événements du cycle de vie d'une activité 191

17. Création de filtres d'intentions 195

18. Lancement d'activités et de sous-activités 201

19. Gestion de la rotation 211

20. Utilisation des ressources 225

21. Utilisation des préférences 245

22. Accès et gestion des bases de données locales 259

23. Accès aux fichiers 273

24. Tirer le meilleur parti des bibliothèques Java 283

25. Communiquer *via* Internet 291

26. Utilisation d'un fournisseur de contenu *(content provider)* 297

27. Construction d'un fournisseur de contenu 305

28. Demander et exiger des permissions 317

29. Création d'un service 323

30. Appel d'un service 331

31. Alerter les utilisateurs avec des notifications 337

32. Accès aux services de localisation 343

33. Cartographie avec *MapView* et *MapActivity* 349

34. Gestion des appels téléphoniques 361

35. Outils de développement 365

36. Gestion des différentes tailles d'écran 377

37. Gestion des terminaux 403

38. Gestion des changements de plateformes . 409

39. Pour aller plus loin 419

Index 423

Table des matières

À propos de l'auteur XVII

Remerciements XIII

Préface à l'édition française XV

Introduction 1

Bienvenue ! 1

Prérequis 1

Éditions de ce livre 2

Termes d'utilisation du code source 2

1. Tour d'horizon 3

Défis de la programmation des smartphones 3

Contenu d'un programme Android 5

Fonctionnalités à votre disposition 6

2. Projets et cibles 7

Les pièces du puzzle 7

Création d'un projet 8

Structure d'un projet 9

Contenu de la racine 9

À la sueur de votre front 9

La suite de l'histoire 10

Le fruit de votre travail 10

Contenu du manifeste 11

Au début, il y avait la racine 11

Permissions, instrumentations et applications 12

Que fait votre application ? 13

Faire le minimum 14

Version = contrôle 15

Émulateurs et cibles 15

Création d'un AVD 15

Choix d'une cible 17

3. Création d'un squelette d'application 19

Terminaux virtuels et cibles 19

Commencer par le début 20

Dissection de l'activité 21

Compiler et lancer l'activité 23

4. Utilisation des layouts XML 27

Qu'est-ce qu'un positionnement XML ? 27

Pourquoi utiliser des layouts XML ? 28
Contenu d'un fichier layout 29
Identifiants des widgets 30
Utilisation des widgets dans le code Java 30
Fin de l'histoire 31

5. Utilisation des widgets de base 33
Labels 33
Boutons 35
Images 35
Champs de saisie 36
Cases à cocher 38
Boutons radio 40
Résumé 42
Propriétés utiles 42
Méthodes utiles 43
Couleurs 43

6. Conteneurs 45
Penser de façon linéaire : *LinearLayout* 46
Concepts et propriétés 46
Exemple 49
Tout est relatif : *RelativeLayout* 53
Concepts et propriétés 53
Exemple 55
Tabula Rasa : *TableLayout* 58
Concepts et propriétés 58
Exemple 60
ScrollView 62

7. Widgets de sélection 65
S'adapter aux circonstances 66
Listes des bons et des méchants 67
Contrôle du Spinner 70
Mettez vos lions en cage 73
Champs : économisez 35 % de la frappe ! 77
Galeries 81

8. S'amuser avec les listes 83
Premières étapes 83
Présentation dynamique 86
Mieux, plus robuste et plus rapide 88
Utilisation de convertView 89
Utilisation du patron de conception "support" 91
Création d'une liste 94
… Et la vérifier deux fois 100
Adapter d'autres adaptateurs 106

9. Utiliser de jolis widgets et de beaux conteneurs 109
Choisir 109
Le temps s'écoule comme un fleuve 114
Mesurer la progression 115
Prendre la bonne résolution 116
Utilisation d'onglets 117
Les pièces du puzzle 117
Idiosyncrasies 118
Code Java 119
Ajouts dynamiques 121
Intent *et* View 124
Tout faire basculer 124
Basculement manuel 126
Ajout de contenu à la volée 127
Basculement automatique 129
Fouiller dans les tiroirs 130
Autres conteneurs intéressants 133

10. Le framework des méthodes de saisie 135
Claviers physiques et logiciels 135

Adaptation à vos besoins 136
Dire à Android où aller 140
Mise en place 142
Libérez le Dvorak qui est en vous 144

11. Utilisation des menus 145
Menus d'options 145
Création d'un menu d'options 146
Ajout de choix et de sous-menus 146
Menus contextuels 147
Illustration rapide 148
Encore de l'inflation 153
Structure XML d'un menu 154
Options des menus et XML 155
Création d'un menu par inflation 156

12. Polices de caractères 157
Sachez apprécier ce que vous avez 157
Polices supplémentaires 159
Le problème des glyphes 161

13. Intégrer le navigateur de WebKit 163
Un navigateur, et en vitesse ! 163
Chargement immédiat 166
Navigation au long cours 167
Amuser le client 167
Réglages, préférences et options 169

14. Affichage de messages surgissants 171
Les toasts 172
Les alertes 172
Mise en œuvre 173

15. Utilisation des threads 177
Les handlers 178
Les messages 178
Les runnables 181
Exécution sur place 181
Où est passé le thread de mon interface utilisateur ? 182
Désynchronisation 182
La théorie 182
AsyncTask, *généricité et paramètres variables* 183
*Les étapes d'*AsyncTask 183
Exemple de tâche 184
Éviter les pièges 188

16. Gestion des événements du cycle de vie d'une activité 191
L'activité de Schroedinger 192
Vie et mort d'une activité 192
onCreate() *et* onDestroy() 192
onStart(), onRestart() *et* onStop() 193
onPause() *et* onResume() 193
L'état de grâce 194

17. Création de filtres d'intentions 195
Quelle est votre intention ? 196
Composantes des intentions 196
Routage des intentions 197
Déclarer vos intentions 198
Récepteurs d'intention 199
Attention à la pause 200

18. Lancement d'activités et de sous-activités 201
Activités paires et sous-activités 202
Démarrage 202
Création d'une intention 203
Faire appel 203
Navigation avec onglets 207

19. Gestion de la rotation 211
Philosophie de la destruction 211
Tout est pareil, juste différent 212
Il n'y a pas de petites économies ! 216
Rotation maison 219
Forcer le destin 221
Tout comprendre 223

20. Utilisation des ressources 225
Les différents types de ressources 225
Théorie des chaînes 226
Chaînes normales 226
Formats de chaînes 227
Texte stylé 227
Formats stylés 228
Vous voulez gagner une image ? 231
Les ressources XML 234
Valeurs diverses 236
Dimensions 237
Couleurs 238
Tableaux 238
Gérer la différence 239

21. Utilisation des préférences 245
Obtenir ce que vous voulez 246
Définir vos préférences 246
Un mot sur le framework 247
Laisser les utilisateurs choisir 247
Ajouter un peu de structure 251
Boîtes de dialogue 253

22. Accès et gestion des bases de données locales 259
Exemple de base de données 260
Présentation rapide de SQLite 261
Commencer par le début 262
Mettre la table 265
Ajouter des données 266
Le retour de vos requêtes 267
Requêtes brutes 267
Requêtes normales 268
Utilisation des "builders" 268
Utilisation des curseurs 270
Des données, des données, encore des données 271

23. Accès aux fichiers 273
Allons-y ! 273
Lire et écrire 277

24. Tirer le meilleur parti des bibliothèques Java 283
Limites extérieures 284
Ant et JAR 284
Suivre le script 285
Tout fonctionne... enfin, presque 289
Autres langages de scripts 289

25. Communiquer *via* Internet 291
REST et relaxation 292
Opérations HTTP via *HttpComponents* ... 292
Traitement des réponses 294
Autres points importants 296

26. Utilisation d'un fournisseur de contenu *(content provider)* 297
Composantes d'une *Uri* 298
Obtention d'un descripteur 298
Création des requêtes 299
S'adapter aux circonstances 300
Insertions et suppressions 302
Attention aux BLOB ! 303

27. Construction d'un fournisseur de contenu 305
D'abord, une petite dissection 306
Puis un peu de saisie 307
Création d'un fournisseur de contenu 307
Étape n° 1 : création d'une classe Provider 307
Étape n° 2 : fournir une Uri 313
Étape n° 3 : déclarer les propriétés 314
Étape n° 4 : modifier le manifeste 314
Avertissements en cas de modifications 315

28. Demander et exiger des permissions 317
Maman, puis-je ? 318
Halte ! Qui va là ? 319
Imposer les permissions via *le manifeste* . 320
Imposer les permissions ailleurs 321
Papiers, s'il vous plaît ! 321

29. Création d'un service 323
Service avec classe 324
Il ne peut en rester qu'un ! 325
Destinée du manifeste 326
Sauter la clôture 327
Méthodes de rappel 327
Intentions diffusées 328

30. Appel d'un service 331
Transmission manuelle 332
Capture de l'intention 334

31. Alerter les utilisateurs avec des notifications 337
Types d'avertissements 337
Notifications matérielles 338
Icônes 338
Les avertissements en action 339

32. Accès aux services de localisation 343
Fournisseurs de localisation : ils savent où vous vous cachez 344
Se trouver soi-même 344
Se déplacer 346
Est-on déjà arrivé ? Est-on déjà arrivé ? Est-on déjà arrivé ? 347
Tester... Tester... 348

33. Cartographie avec *MapView* et *MapActivity* 349
Termes d'utilisation 350
Empilements 350
Les composants essentiels 351
Testez votre contrôle 352
Zoom 352
Centrage 353
Terrain accidenté 353
Couches sur couches 354
Classes Overlay 354
*Affichage d'*ItemizedOverlay 354
Gestion de l'écran tactile 356
Moi et MyLocationOverlay 357
La clé de tout 358

34. Gestion des appels téléphoniques 361
Le Manager 362
Appeler 362

35. Outils de développement 365
Gestion hiérarchique 365
DDMS (*Dalvik Debug Monitor Service*) 370
Journaux 371
Stockage et extraction de fichiers 371
Copies d'écran 372

Mise à jour de la position 373
Appels téléphoniques et SMS 374
Gestion des cartes amovibles 375
Création d'une image de carte 376

36. Gestion des différentes tailles d'écran 377
Écran par défaut 378
Tout en un 378
Penser en termes de règles, pas en termes de positions 379
Utilisez des dimensions physiques 380
Évitez les vrais pixels 380
Choisir des images adaptables 380
Fait maison, rien que pour vous... 381
<supports-screens> 381
Ressources et ensembles de ressources 382
Trouver sa taille 383
Rien ne vaut la réalité 384
Différences de densité 384
Ajustement de la densité 385
Accès aux véritables terminaux 386
Exploitez sans vergogne la situation 386
Remplacer les menus par des boutons 386
Remplacer les onglets par une seule activité 387
Consolider les activités multiples 387
Exemple : *EU4You* 388
Première version 388
Corriger les polices 395
Corriger les icônes 397
Utilisation de l'espace 397
Et si ce n'est pas un navigateur ? 400
Nos amis ont-ils quelques bogues ? 401

37. Gestion des terminaux 403
Cette application contient des instructions explicites 403
Boutons, boutons, qui a des boutons ? 405
Un marché garanti 405
Les détails scabreux 406
Tablette Internet Android Archos 5 406
CLIQ/DEXT de Motorola 406
DROID/Milestone de Motorola 407
Nexus One de Google/HTC 407
BACKFLIP de Motorola 407

38. Gestion des changements de plateformes 409
Gestion de la marque 410
Des choses qui risquent de vous rendre nerveux 410
Hiérarchie des vues 410
Changement des ressources 411
Gestion des modifications de l'API 412
Détecter la version d'Android 412
Envelopper l'API 412

39. Pour aller plus loin 419
Questions avec, parfois, des réponses 419
Aller à la source 420
Lire les journaux 421

Index 423

À propos de l'auteur

Mark Murphy est le fondateur de CommonsWare et l'auteur de *The Busy Coder's Guide to Android Development.* Son expérience professionnelle va du conseil pour les développements open-source et collaboratifs de très grosses sociétés au développement d'applications sur à peu près tout ce qui est plus petit qu'un mainframe. Il programme depuis plus de vingt-cinq ans et a travaillé sur des plates-formes allant du TRS-80 aux derniers modèles de terminaux mobiles. En tant qu'orateur averti, Mark intervient également sur un grand nombre de sujets dans de nombreuses conférences et sessions de formation internationales.

Par ailleurs, Mark est le rédacteur des rubriques "Building Droids" d'AndroidGuys et "Android Angle" de NetworkWorld.

En dehors de CommonsWare, Mark s'intéresse beaucoup au rôle que joue Internet dans l'implication des citoyens dans la politique. À ce titre, il publie des articles dans la collection Rebooting America et son blog personnel contient de nombreux articles sur la "démocratie coopérative".

Remerciements

Je voudrais remercier l'équipe d'Android ; non seulement pour avoir créé un bon produit, mais également pour l'aide inestimable qu'elle fournit dans les groupes Google consacrés à ce système. Merci notamment à Romain Guy, Justin Mattson, Dianne Hackborn, Jean-Baptiste Queru, Jeff Sharkey et Xavier Ducrohet.

Les icônes utilisées dans les exemples de ce livre proviennent du jeu d'icônes Nuvola[1].

1. **http://www.icon-king.com/?p=15**.

Préface à l'édition française

Novembre 2007, Google, le géant américain de l'Internet annonce qu'il vient de créer un nouveau système d'exploitation pour appareil mobile. Nommé Android, ce système est totalement gratuit et open-source, pour les développeurs d'applications, mais également pour les constructeurs d'appareils mobiles (téléphones, smartphones, MID [*Multimedia Interface Device*], GPS...).

Un an auparavant, Apple avait lancé un pavé dans la mare en s'introduisant avec brio dans le marché des systèmes d'exploitation mobiles, détenus à plus de 50 % par Symbian, grâce à l'apport de nombreuses fonctionnalités multimédia et de localisation dans une interface utilisateur très intuitive.

Android apporte également son lot de nouvelles fonctionnalités, mais avec des atouts supplémentaires : gratuité, code ouvert, langage de programmation approuvé et connu de millions de développeurs à travers le monde. Autour d'une alliance d'une quarantaine de constructeurs, éditeurs logiciels et sociétés spécialisées dans les applications mobiles (Open Handset Alliance), Google a de quoi inquiéter les systèmes d'exploitation propriétaires tels que Windows, Palm, Samsung, RIM, Symbian.

Certains développeurs ont immédiatement cru à Android, dès son annonce par Google, et des blogs, des sites Internet et des tutoriaux ont rapidement vu le jour de par le monde. Ce phénomène a été encouragé par un concours de développement à l'initiative de Google – avec plus de 1 million de dollars de lots. Le but était bien entendu de faire parler de son nouveau système : pari réussi ! En 4 mois, alors que la documentation de la plateforme était quasiment inexistante, les juges du concours ont reçu pas moins de 1 700 applications !

Ceux qui ont cru en Android ne se sont pas trompés sur son potentiel ni sur celui des nouveaux services mobiles émergents ! Un an plus tard, tous les grands constructeurs de téléphonie mobile annonçaient un ou plusieurs smartphones équipés de cette plateforme, sans compter sur l'arrivée de nouveaux constructeurs. Aujourd'hui, le portail de vente de Google dédié aux applications Android (Android Market) dispose d'environ 80 000 applications, et 70 000 nouveaux appareils fonctionnant sur Android sont activés chaque jour. Internet relate quotidiennement l'arrivée de smartphones Android, mais aussi de tablettes internet, autoradios, téléviseurs, box internet... Tous les constructeurs de matériels touchant de près ou de loin à l'information travaillent ou travailleront prochainement avec Android.

Mark Murphy, auteur du présent ouvrage, est l'un de ces développeurs qui a cru à Android dès ses premières heures. Présent sur les groupes de discussions Google, il apportait son aide aux programmeurs en quête d'informations. C'est d'ailleurs dans ce contexte que j'ai fait sa connaissance. Mark s'est attaché, dans cette seconde édition de *L'Art du développement Android*, à prendre en compte les dernières mises à jour du système d'exploitation : Android a sorti 4 versions sur l'année 2009, et 3 sont prévues au total sur 2010... Dans sa dernière annonce, l'équipe Android a indiqué qu'il n'y aurait plus qu'une version par an à partir de 2011 !

Cet ouvrage sera donc une très bonne base pour vos développements actuels et à venir. Toutes les bases de la programmation sous Android y sont décrites, mais pas uniquement. Cet ouvrage regorge d'astuces et de conseils qui vous permettront de pouvoir réaliser des applications Android convaincantes.

Arnaud Farine
eXpertise @ndroid
http://www.expertiseandroid.com/

Introduction

Bienvenue !

Merci de votre intérêt pour le développement d'applications Android ! De plus en plus de personnes accèdent désormais aux services Internet *via* des moyens "non traditionnels" comme les terminaux mobiles, et ce nombre ne peut que croître. Bien qu'Android soit récent – ses premiers terminaux sont apparus à la fin de 2008 –, il ne fait aucun doute qu'il se répandra rapidement grâce à l'influence et à l'importance de l'*Open Handset Alliance*.

Merci surtout de votre intérêt pour ce livre ! J'espère sincèrement que vous le trouverez utile, voire divertissant par moments.

Prérequis

Pour programmer des applications Android, vous devez au moins connaître les bases de Java. En effet, la programmation Android utilise la syntaxe de ce langage, plus une bibliothèque de classes s'apparentant à un sous-ensemble de la bibliothèque de Java SE (avec des extensions spécifiques). Si vous n'avez jamais programmé en Java, initiez-vous à ce langage avant de vous plonger dans la programmation Android.

Ce livre n'explique pas comment télécharger ou installer les outils de développement Android, que ce soit le plugin Eclipse ou les outils dédiés. Vous trouverez toutes ces informations sur le site web d'Android[1]. Tout ce qui est expliqué dans cet ouvrage ne tient pas compte de l'environnement que vous utilisez et fonctionnera dans tous les cas. Nous vous conseillons de télécharger, d'installer et de tester ces outils de développement avant d'essayer les exemples de ce livre.

1. **http://code.google.com/android/index.html**.

Éditions de ce livre

Cet ouvrage est le fruit d'une collaboration entre Apress et CommonsWare. La version que vous êtes en train de lire est l'édition d'Apress, qui est disponible sous forme imprimée ou numérique.

De son côté, CommonsWare met continuellement à jour le contenu original et le met à disposition des membres de son programme Warescription sous le titre *The Busy Coder's Guide to Android Development*.

La page **http://commonsware.com/apress** contient une FAQ concernant ce partenariat avec Apress.

Termes d'utilisation du code source

Le code source des exemples de ce livre est disponible à partir du site web de Pearson (**www.pearson.fr**), sur la page consacrée à cet ouvrage. Tous les projets Android sont placés sous les termes de la licence Apache 2.0[1], que nous vous invitons à lire avant de réutiliser ces codes.

1. **http://www.apache.org/licenses/LICENSE-2.0.html**.

1

Tour d'horizon

Les terminaux Android seront essentiellement des téléphones mobiles, même si de plus en plus de tablettes sont annoncées.

Défis de la programmation des smartphones

Du côté des avantages, les téléphones Android qui arrivent sur le marché sont assez jolis. Si l'offre de services Internet pour les mobiles remonte au milieu des années 1990, avec HDML (*Handheld Device Markup Language*), ce n'est que depuis ces dernières années que l'on dispose de téléphones capables d'exploiter pleinement l'accès à Internet. Grâce aux SMS et à des produits comme l'iPhone d'Apple, les téléphones qui peuvent servir de terminaux Internet deviennent de plus en plus courants. Travailler sur des applications Android permet donc d'acquérir une expérience non négligeable dans une technologie moderne (Android) et dans un segment de marché qui croît rapidement (celui des terminaux mobiles pour Internet).

Le problème intervient lorsqu'il s'agit de programmer. Quiconque a déjà écrit des programmes pour des PDA (*Personal Digital Assistant*) ou des téléphones portables s'est heurté aux inconvénients de leur miniaturisation :

- Les écrans sont sous-dimensionnés.
- Les claviers, quand ils existent, sont minuscules.
- Les dispositifs de pointage, quand il y en a, sont peu pratiques (qui n'a pas déjà perdu son stylet ?) ou imprécis (les gros doigts se marient mal avec les écrans tactiles).
- La vitesse du processeur et la taille de la mémoire sont limitées par rapport à celles des machines de bureau et des serveurs auxquels nous sommes habitués.

On peut utiliser le langage de programmation et le framework de développement que l'on souhaite, à condition que ce soit celui qu'a choisi le constructeur du terminal.En outre, les applications qui s'exécutent sur un téléphone portable doivent gérer le fait qu'il s'agit justement *d'un téléphone*.

Les utilisateurs sont généralement assez irrités lorsque leur mobile ne fonctionne pas, et c'est la raison pour laquelle la campagne publicitaire "Can you hear me now?" de Verizon a si bien fonctionné ces dernières années. De même, ces mêmes personnes seront très mécontentes si votre programme perturbe leur téléphone, pour les raisons suivantes :

- Il occupe tellement le processeur qu'elles ne peuvent plus recevoir d'appels.
- En ne s'effaçant pas à l'arrière-plan lorsque cela est nécessaire, parce qu'il ne s'intègre pas correctement au système d'exploitation de leur mobile..
- Il provoque un plantage de leur téléphone à cause d'une fuite de mémoire.

Le développement de programmes pour un téléphone portable est donc différent de l'écriture d'applications pour des machines de bureau, du développement de sites web ou de la création de programmes serveurs. Vous finirez par utiliser des outils et des frameworks différents et vos programmes auront des limites auxquelles vous n'êtes pas habitué.

Android essaie de vous faciliter les choses :

- Il fournit un langage de programmation connu (Java), avec des bibliothèques relativement classiques (certaines API d'Apache, par exemple), ainsi qu'un support pour les outils auxquels vous êtes peut-être habitué (Eclipse, notamment).
- Il vous offre un framework suffisamment rigide et étanche pour que vos programmes s'exécutent "correctement" sur le téléphone, sans interférer avec les autres applications ou le système d'exploitation lui-même.

Comme vous vous en doutez sûrement, l'essentiel de ce livre s'intéresse à ce framework et à la façon d'écrire des programmes qui fonctionnent dans son cadre et tirent parti de ses possibilités.

Contenu d'un programme Android

Le développeur d'une application classique est "le seul maître à bord". Il peut ouvrir la fenêtre principale de son programme, ses fenêtres filles – les boîtes de dialogue, par exemple – comme il le souhaite. De son point de vue, il est seul au monde ; il tire parti des fonctionnalités du système d'exploitation mais ne s'occupe pas des autres programmes susceptibles de s'exécuter en même temps que son programme. S'il doit interagir avec d'autres applications, il passe généralement par une API, comme JDBC (*Java DataBase Connectivity*) ou les frameworks qui reposent sur lu pour communiquer avec MySQL ou un autre SGBDR.

Android utilise les mêmes concepts, mais proposés de façon différente, avec une structure permettant de mieux protéger le fonctionnement des téléphones. Les composants principaux d'une application Android sont :

- **Les activités (*activities*).** Ce sont les briques de base de l'interface utilisateur. Vous pouvez considérer une activité comme l'équivalent Android de la fenêtre ou de la boîte de dialogue d'une application classique. Bien que des activités puissent ne pas avoir d'interface utilisateur, un code "invisible" sera délivré le plus souvent sous la forme de fournisseurs de contenus (content provider) ou de services.
- **Les fournisseurs de contenus (*content providers*).** Ils offrent un niveau d'abstraction pour toutes les données stockées sur le terminal et accessibles aux différentes applications. Le modèle de développement Android encourage la mise à disposition de ses propres données aux autres programmes – construire un fournisseur de contenus permet d'obtenir ce résultat tout en gardant un contrôle total sur la façon dont on accédera aux données.
- **Les services.** Les activités et les fournisseurs de contenus ont une durée de vie limitée et peuvent être éteints à tout moment. Les services sont en revanche conçus pour durer et, si nécessaire, indépendamment de toute activité. Vous pouvez, par exemple, utiliser un service pour vérifier les mises à jour d'un flux RSS ou pour jouer de la musique, même si l'activité de contrôle n'est plus en cours d'exécution.
- **Les intentions (*intents*).** Ce sont des messages système émis par le terminal pour prévenir les applications de la survenue de différents événements, que ce soit une modification matérielle (comme l'insertion d'une carte SD) ou l'arrivée de données (telle la réception d'un SMS), en passant par les événements des applications elles-mêmes (votre activité a été lancée à partir du menu principal du terminal, par exemple). Vous pouvez non seulement répondre aux intentions, mais également créer les vôtres afin de lancer d'autres activités ou pour vous prévenir qu'une situation particulière a lieu (vous pouvez, par exemple, émettre l'intention X lorsque l'utilisateur est à moins de 100 mètres d'un emplacement Y).

Fonctionnalités à votre disposition

Android fournit un certain nombre de fonctionnalités pour vous aider à développer des applications :

- **Stockage.** Vous pouvez assembler des fichiers de données dans une application, pour y stocker ce qui ne changera jamais – les icônes ou les fichiers d'aide, par exemple. Vous pouvez également réserver un petit emplacement sur le terminal lui-même, pour y stocker une base de données ou des fichiers contenant des informations nécessaires à votre application et saisies par l'utilisateur ou récupérées à partir d'une autre source. Si l'utilisateur fournit un espace de stockage comme une carte SD, celui-ci peut également être lu et écrit en fonction des besoins.
- **Réseau.** Les terminaux Android sont généralement conçus pour être utilisés avec Internet, *via* un support de communication quelconque. Vous pouvez tirer parti de cet accès à Internet à n'importe quel niveau, des sockets brutes de Java à un widget de navigateur web intégré que vous pouvez intégrer dans votre application.
- **Multimédia.** Les terminaux Android permettent d'enregistrer et de jouer de la musique et de la vidéo. Bien que les caractéristiques spécifiques varient en fonction des modèles, vous pouvez connaître celles qui sont disponibles et tirer parti des fonctionnalités multimédias offertes, que ce soit pour écouter de la musique, prendre des photos ou enregistrer des mémos vocaux.
- **GPS (*Global Positioning System*).** Les fournisseurs de positionnement, comme GPS, permettent d'indiquer aux applications où se trouve le terminal. Il vous est alors possible d'afficher des cartes ou d'utiliser ces données géographiques pour retrouver la trace du terminal s'il a été volé, par exemple.
- **Services téléphoniques.** Les terminaux Android sont, bien sûr, généralement des téléphones, ce qui permet à vos programmes de passer des appels, d'envoyer et de recevoir des SMS (*Short Message Service*) et de réaliser tout ce que vous êtes en droit d'attendre d'une technologie téléphonique moderne.

2

Projets et cibles

Après avoir téléchargé et installé le dernier kit de développement Android (SDK) et, éventuellement, l'extension ADT (*Android Developper Tools*) d'Eclipse, vous êtes prêt à travailler. Ce chapitre présente tous les composants impliqués dans la construction d'une application Android.

Les pièces du puzzle

Pour créer une application Android, vous devez créer un projet, qui peut être un projet Eclipse si vous utilisez cet environnement. Ce projet contiendra tout le code source, les ressources (les messages traduits dans différentes langues, par exemple), les JAR tiers et tout ce qui est associé à l'application. Les outils Android, qu'ils soient intégrés à Eclipse ou non, se chargeront ensuite de transformer le contenu de ce projet en un fichier paquetage Android (APK) représentant l'application elle-même. Ces outils vous aideront également à placer ce fichier APK sur un émulateur ou sur un véritable terminal Android afin de le tester.

L'un des éléments essentiels d'un projet Android est son *manifeste* (`AndroidManifest.xml`). Ce fichier contient en effet la "table des matières" de l'application, il énumère ses composants essentiels, il précise ses permissions, etc. Android s'en sert au lancement de l'application pour intégrer celle-ci au système d'exploitation. Le contenu du manifeste est également utilisé par l'Android Market (et, éventuellement, par d'autres "app stores" indépendantes) pour que les applications qui requièrent Android 2.0 ne soient proposées à ceux qui ont des terminaux Android 1.5, par exemple.

Pour tester votre application avec l'émulateur, vous devrez créer un AVD (Android Virtual Device). Généralement, on en crée plusieurs afin d'avoir un terminal virtuel par configuration matérielle. Vous pouvez donc avoir des AVD pour des tailles d'écran différentes, des versions d'Android différentes, etc.

Lorsque vous créez des projets et des AVD, vous devez indiquer à Android le niveau de l'API avec laquelle vous travaillez. Ce niveau est un simple entier correspondant à une version d'Android : le niveau 3, par exemple, correspond à Android 1.5. Dans le cas d'un projet, vous pourrez indiquer à Android les niveaux minimum et maximum reconnus par votre application ; dans le cas d'un AVD, vous pourrez signaler le niveau d'API à émuler : vous saurez ainsi comment se comporte votre application sur différents (faux) terminaux utilisant des versions différentes d'Android.

Ce chapitre décrit en détail tous ces concepts.

Création d'un projet

`android create project` permet de créer un projet à partir de la ligne de commande afin de l'utiliser ensuite avec les outils comme `ant`. Cette commande reconnaît un certain nombre d'options permettant d'indiquer le paquetage Java dans lequel sera placé le code de l'application, le niveau voulu pour l'application, etc. Son exécution produit un répertoire contenant tous les fichiers nécessaires à la création d'une application Android "Hello, World !".

Voici un exemple d'utilisation :

```
android create project --target 2 --path ./FirstApp <FL>
                    ➥--activity FirstApp --package apt.tutorial
```

Si vous comptez vous servir d'Eclipse pour vos développements Android, vous utiliserez plutôt l'assistant de création de projets.

Le code source qui accompagne cet ouvrage a été conçu pour être compilé avec les outils en ligne de commande. Si vous préférez utiliser Eclipse, vous pouvez créer des projets Android vides sous Eclipse, puis y importer le code.

Structure d'un projet

Le système de construction d'un programme Android est organisé sous la forme d'une arborescence de répertoires spécifique à un projet, exactement comme n'importe quel projet Java. Les détails, cependant, sont spécifiques à Android. Voici un rapide tour d'horizon de la structure d'un projet, qui vous aidera à mieux comprendre les exemples de code utilisés dans ce livre et que vous pouvez télécharger sur le site web de Pearson, **www.pearson.fr**, sur la page consacrée à cet ouvrage.

Contenu de la racine

Lorsque vous créez un projet Android (avec `android create project`, par exemple), plusieurs éléments sont placés dans le répertoire racine du projet. Parmi eux :

- `AndroidManifest.xml` est un fichier XML qui décrit l'application à construire et les composants – activités, services, etc. – fournis par celle-ci.
- `build.xml` est un script Ant[1] permettant de compiler l'application et de l'installer sur le terminal.
- `default.properties` et `local.properties` sont des fichiers de propriétés utilisés par le script précédent.
- `assets/` contient les autres fichiers statiques fournis avec l'application pour son déploiement sur le terminal.
- `bin/` contient l'application compilée.
- `gen/` contient le code source produit par les outils de compilation d'Android.
- `libs/` contient les fichiers JAR extérieurs nécessaires à l'application.
- `src/` contient le code source Java de l'application.
- `res/` contient les ressources – icônes, descriptions des éléments de l'interface graphique (*layouts*), etc. – assemblées avec le code Java compilé.
- `tests/` contient un projet Android entièrement distinct, utilisé pour tester celui que vous avez créé.

À la sueur de votre front

Lors de la création d'un projet Android (avec `android create project`, par exemple), vous devez fournir le nom de classe ainsi que le chemin complet (paquetage) de l'activité

1. **http://ant.apache.org/**.

"principale" de l'application (`com.commonsware.android.UneDemo`, par exemple). Vous constaterez alors que l'arborescence `src/` de ce projet contient la hiérarchie des répertoires définis par le paquetage ainsi qu'un squelette d'une sous-classe d'`Activity` représentant l'activité principale (`src/com/commonsware/android/UneDemo.java`). Vous pouvez bien sûr modifier ce fichier et en ajouter d'autres à l'arborescence `src/` selon les besoins de votre application.

La première fois que vous compilerez le projet (avec `ant`, par exemple), la chaîne de production d'Android créera le fichier `R.java` dans le paquetage de l'activité "principale". Ce fichier contient un certain nombre de définitions de constantes liées aux différentes ressources de l'arborescence `res/`. Vous rencontrerez de nombreuses références à `R.java` dans les exemples de ce livre (par exemple, on désignera l'identifiant d'un layout par `R.layout.main`).

Ne modifiez pas `R.java` vous-même, laissez les outils Android le faire pour vous.

La suite de l'histoire

Comme on l'a déjà indiqué, l'arborescence `res/` contient les ressources, c'est-à-dire des fichiers statiques fournis avec l'application, soit sous leur forme initiale soit, parfois, sous une forme prétraitée. Parmi les sous-répertoires de `res/`, citons :

- `res/drawable/` pour les images (PNG, JPEG, etc.) ;
- `res/layout/` pour les descriptions XML de la composition de l'interface graphique ;
- `res/menu/` pour les descriptions XML des menus ;
- `res/raw/` pour les fichiers généraux (un fichier CSV contenant les informations d'un compte, par exemple) ;
- `res/values/` pour les messages, les dimensions, etc. ;
- `res/xml/` pour les autres fichiers XML généraux que vous souhaitez fournir.

Nous présenterons tous ces répertoires, et bien d'autres, dans la suite de ce livre.

Le fruit de votre travail

Lorsque vous compilez un projet (avec `ant` ou un IDE), le résultat est placé dans le répertoire `bin/`, sous la racine de l'arborescence du projet :

- `bin/classes/` contient les classes Java compilées.
- `bin/classes.dex` contient l'exécutable créé à partir de ces classes compilées.

- `bin/votreapp.ap_` (où `votreapp` est le nom de l'application) contient les ressources de celle-ci, sous la forme d'un fichier ZIP.
- `bin/votreapp-debug.apk` ou `bin/votreapp-unsigned.apk` est la véritable application Android.

Le fichier `.apk` est une archive ZIP contenant le fichier `.dex`, la version compilée des ressources (`resources.arsc`), les éventuelles ressources non compilées (celles qui se trouvent sous `res/raw/`, par exemple) et le fichier `AndroidManifest.xml`. Cette archive est signée : la partie `-debug` du nom de fichier indique qu'elle l'a été à l'aide d'une clé de débogage qui fonctionne avec l'émulateur alors que `-unsigned` précise que l'application a été construite pour être déployée (`ant release`) : l'archive APK doit alors être signée à l'aide de `jarsigner` et d'une clé officielle.

Contenu du manifeste

Le point de départ de toute application Android est son fichier manifeste, `AndroidManifest.xml`, qui se trouve à la racine du projet. C'est dans ce fichier que l'on déclare ce que contiendra l'application – les activités, les services, etc. On y indique également la façon dont ces composants seront reliés au système Android lui-même en précisant, par exemple, l'activité (ou les activités) qui doivent apparaître dans le menu principal du terminal (ce menu est également appelé "lanceur" ou "launcher").

La création d'une application produit automatiquement un manifeste de base. Pour une application simple, qui offre uniquement une activité, ce manifeste automatique conviendra sûrement ou nécessitera éventuellement quelques modifications mineures. En revanche, le manifeste de l'application de démonstration Android fournie contient plus de 1 000 lignes. Vos applications se situeront probablement entre ces deux extrémités.

La plupart des parties intéressantes du manifeste seront décrites en détail dans les chapitres consacrés aux fonctionnalités d'Android qui leur sont associées – l'élément `service`, par exemple, est détaillé au Chapitre 29. Pour le moment, il vous suffit de comprendre le rôle de ce fichier et la façon dont il est construit.

Au début, il y avait la racine

Tous les fichiers manifestes ont pour racine un élément `manifest` :

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.search">
...
</manifest>
```

Cet élément comprend la déclaration de l'espace de noms `android`. Curieusement, les manifestes produits n'appliquent cet espace qu'aux attributs, pas aux éléments (on écrit donc `manifest`, et non `android:manifest`). Nous vous conseillons de conserver cette convention, sauf si Android la modifie dans une version future.

L'essentiel des informations que vous devez fournir à cet élément est l'attribut `package` (qui n'utilise pas non plus l'espace de noms). C'est là que vous pouvez indiquer le nom du paquetage Java qui sera considéré comme la "base" de votre application. Dans la suite du fichier, vous pourrez alors simplement utiliser le symbole point pour désigner ce paquetage : si vous devez, par exemple, faire référence à `com.commonsware.android.search.Snicklefritz` dans le manifeste de cet exemple, il suffira d'écrire `.Snicklefritz` puisque `com.commonsware.android.search` est défini comme le paquetage de l'application.

Permissions, instrumentations et applications

Sous l'élément `manifest`, vous trouverez les éléments suivants :

- Des éléments `uses-permission` indiquant les permissions dont a besoin votre application pour fonctionner correctement.
- Des éléments `permission` déclarant les permissions que les activités ou les services peuvent exiger des autres applications pour utiliser les données et le code de l'application.
- Des éléments `instrumentation` qui indiquent le code qui devrait être appelé pour les événements système essentiels, comme le lancement des activités. Ces éléments sont utilisés pour la journalisation ou la surveillance.
- Des éléments `uses-library` pour relier les composants facultatifs d'Android, comme les services de géolocalisation.
- Un élément `uses-sdk` indiquant la version du SDK Android avec laquelle a été construite l'application.
- Un élément `application` qui définit le cœur de l'application décrite par le manifeste.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android">
  <uses-permission
    android:name="android.permission.ACCESS_LOCATION" />
  <uses-permission
    android:name="android.permission.ACCESS_GPS" />
  <uses-permission
    android:name="android.permission.ACCESS_ASSISTED_GPS" />
  <uses-permission
```

```
    android:name="android.permission.ACCESS_CELL_ID" />
  <application>
...
  </application>
</manifest>
```

Le manifeste de cet exemple contient des éléments `uses-permission` qui indiquent certaines fonctionnalités du terminal dont l'application a besoin – ici, l'application doit avoir le droit de déterminer sa position géographique courante, par exemple. L'élément `application` décrit les activités, les services et tout ce qui constitue l'application elle-même.

Les permissions seront décrites en détail au Chapitre 28.

Que fait votre application ?

Le plat de résistance du fichier manifeste est décrit par les fils de l'élément `application`.

Par défaut, la création d'un nouveau projet Android n'indique qu'un seul élément `activity` :

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.commonsware.android.skeleton">
  <application>
    <activity android:name=".Now" android:label="Now">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Cet élément fournit `android:name` pour la classe qui implémente l'activity, `android:label` pour le nom affiché de l'activity et (souvent) un élément fils `intent-filter` décrivant les conditions sous lesquelles cette activité s'affichera. L'élément `activity` de base configure votre activité pour qu'elle apparaisse dans le lanceur sous forme d'une icône et que les utilisateurs puissent l'exécuter. Comme nous le verrons plus tard, un même projet peut définir plusieurs activités.

Il peut également y avoir un ou plusieurs éléments `provider` indiquant les fournisseurs de contenus (*content providers*) – les composants qui fournissent les données à vos activités et, avec votre permission, aux activités d'autres applications du terminal. Ces éléments encapsulent les bases de données ou les autres stockages de données en une API unique que toute application peut ensuite utiliser. Nous verrons plus loin comment créer des fournisseurs de contenus et comment utiliser les fournisseurs que vous ou d'autres ont créés.

Enfin, il peut y avoir un ou plusieurs éléments `service` décrivant les services, c'est-à-dire les parties de code qui peuvent fonctionner indépendamment de toute activité et en permanence. L'exemple classique est celui du lecteur MP3, qui permet de continuer à écouter de la musique, même si l'utilisateur ouvre d'autres activités et que l'interface utilisateur n'est pas affichée au premier plan. La création et l'utilisation des services sont décrites aux Chapitres 29 et 30.

Faire le minimum

Android, comme la plupart des systèmes d'exploitation, est régulièrement amélioré, ce qui donne lieu à de nouvelles versions du système. Certains de ces changements affectent le SDK car de nouveaux paramètres, classes ou méthodes apparaissent, qui n'existaient pas dans les versions précédentes.

Pour être sûr que votre application ne s'exécutera que pour une version précise (ou supérieure) d'un environnement Android, ajoutez un élément `uses-sdk` comme fils de l'élément racine `manifest` du fichier `AndroidManifest.xml`. `uses-sdk` dispose de l'attribut `minSdkVersion`, indiquant la version minimale du SDK exigée par votre application :

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.commonsware.android.search">
  <uses-sdk minSdkVersion="2" />
  ...
</manifest>
```

À l'heure où ce livre est écrit, `minSdkVersion` peut prendre sept valeurs :

- 1 pour indiquer le premier SDK d'Android, la version 1.0 ;
- 2 pour indiquer la version 1.1 du SDK d'Android ;
- 3 pour indiquer la version 1.5 ;
- 4 pour indiquer la version 1.6 ;
- 5 pour indiquer la version 2.0 ;
- 6 pour indiquer la version 2.0.1 ;
- 7 pour indiquer la version 2.1.

L'absence d'un élément `uses-sdk` revient à utiliser 1 comme valeur de `minSdkVersion`.

*La boutique Android (*Android Market*) semble tenir à ce que l'on indique explicitement la valeur de* `minSdkVersion` *: faites en sorte d'utiliser un élément* `uses-sdk` *si vous comptez distribuer votre programme par ce biais.*

Si cet élément est présent, l'application ne pourra s'installer que sur les terminaux compatibles. Vous n'avez pas besoin de préciser la dernière version du SDK mais, si vous en choisissez une plus ancienne, c'est à vous de vérifier que votre application fonctionnera sur toutes les versions avec lesquelles elle prétend être compatible. Si, par exemple, vous omettez `uses-sdk`, cela revient à annoncer que votre application fonctionnera sur toutes les versions existantes du SDK et vous devrez évidemment tester que c'est bien le cas.

Version = contrôle

Si vous comptez distribuer votre application *via* la boutique Android ou d'autres supports, vous devrez sûrement ajouter les attributs `android:versionCode` et `android:versionName` à l'élément `manifest` afin de faciliter le processus de mise à jour des applications.

- `android:versionName` est une chaîne lisible représentant le nom ou le numéro de version de votre application. Vous pouvez, par exemple, utiliser des valeurs comme "3.0", "System V", "5000" ou "3.1" en fonction de vos préférences.
- `android:versionCode` est un entier censé représenter le numéro de version de l'application. Le système l'utilise pour savoir si votre version est plus récente qu'une autre – "plus récent" étant défini par "la valeur d'android:versionCode est plus élevée". Pour obtenir cette valeur, vous pouvez convertir le contenu d'android:versionName en nombre ou simplement incrémenter la valeur à chaque nouvelle version.

Émulateurs et cibles

Prenons un moment pour présenter la notion de *cibles* Android car elle est parfois mal comprise. Les cibles sont importantes pour le développement des applications sur le long terme, notamment lorsque l'on utilise l'émulateur pour les tester.

Création d'un AVD

Pour utiliser l'émulateur, vous devez créer un ou plusieurs AVD. Ces terminaux virtuels sont conçus pour imiter les véritables terminaux Android comme le G1 ou le HTC Magic. Il suffit d'indiquer l'AVD que l'on utilise et l'émulateur prétendra qu'il est le terminal décrit par celui-ci.

Vous devez indiquer une cible lorsque vous créez un AVD. Cette cible indique la classe de terminaux que l'AVD émulera. À l'heure où ce livre est écrit, il existe sept cibles :

- 1 : terminal Android 1.1, comme un G1 non mis à jour ;
- 2 : terminal Android 1.5, sans le support de Google Maps (ce qui peut être le cas d'une version Android fournie pour un terminal particulier) ;

- 3 : terminal Android 1.5 disposant du support de Google Maps ;
- 4 : terminal Android 1.6 disposant du support de Google Maps ;
- 5 : terminal Android 2.0 disposant du support de Google Maps ;
- 6 : terminal Android 2.0.1 disposant du support de Google Maps ;
- 7 : terminal Android 2.1 disposant du support de Google Maps.

La commande `android list targets` *vous donnera la listes des API disponibles.*

Si vous développez des applications utilisant Google Maps, vous devez donc utiliser un AVD ayant une cible supérieure ou égale à 3.

Vous pouvez créer autant d'AVD que vous le souhaitez du moment que vous avez assez d'espace disque disponible sur votre environnement de développement. Chaque AVD se comporte comme un terminal totalement distinct : l'installation d'une application sur un AVD n'affecte pas les autres AVD que vous avez créés.

Pour créer un AVD, utilisez la commande `android create avd`, Eclipse ou le gestionnaire d'AVD (*AVD Manager*), un outil graphique qui a été ajouté au SDK d'Android 1.6. Pour utiliser ce dernier, lancez simplement la commande `android` sans aucun paramètre. Comme le montre la Figure 2.1, le gestionnaire affiche une liste d'AVD prédéfinis. Les boutons `New` et `Delete` permettent, respectivement, d'ajouter et de supprimer des AVD ; le bouton `Start` lance un émulateur avec l'AVD sélectionné, etc.

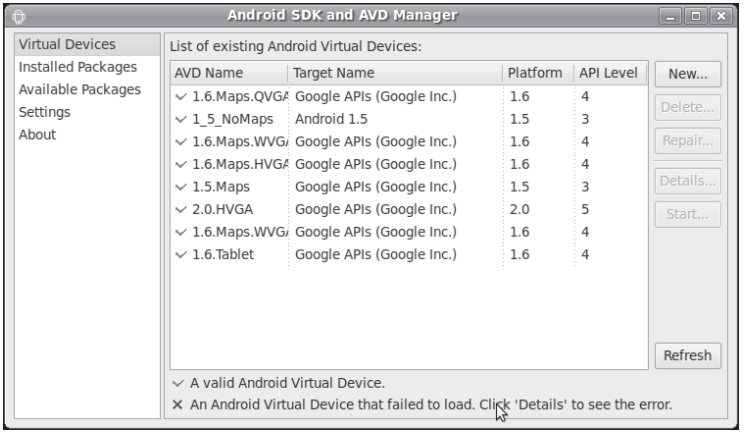
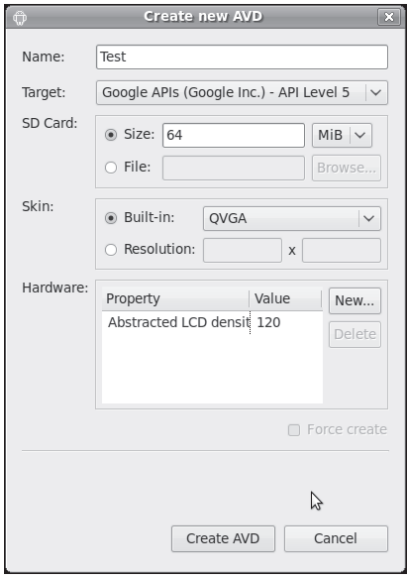


Figure 2.1
L'outil graphique AVD Manager, montrant la liste des AVD disponibles.

Pour ajouter un AVD avec l'interface graphique et son bouton New, il faut fournir un nom, une combinaison API/Google, des détails sur l'image d'une carte SD et la taille de l'écran (le skin) que l'on veut émuler. La Figure 2.2 montre la boîte de dialogue de création d'un AVD.

Figure 2.2
Ajout d'un AVD.



Choix d'une cible

Lorsque vous créez un projet (avec `android create project` ou à partir d'Eclipse), vous devez également indiquer la classe de terminal visée par celui-ci. Les valeurs possibles étant les mêmes que ci-dessus, créer un projet avec une cible 3 donne les indications suivantes :

- Vous avez besoin d'Android 1.5.
- Vous avez besoin de Google Maps.

L'application finale ne s'installera donc pas sur les terminaux qui ne correspondent pas au minimum à ces critères.

Voici quelques règles pour vous aider à gérer les cibles :

- Demandez-vous ce dont vous avez réellement besoin. Si vous vous en tenez aux API 1.5, ne compilez votre application que pour Android 1.5.
- Testez autant de cibles que possible. Vous pourriez être tenté par la cible 1 pour viser le plus grand nombre de terminaux Android ; cependant, vous devrez alors tester votre

application sur un AVD ayant une cible 1 et un AVD ayant une cible 2 (et il serait également souhaitable de la tester avec un AVD ayant une cible 3, au cas où).

- Vérifiez qu'une nouvelle cible n'a pas été ajoutée par une nouvelle version d'Android. Il devrait y avoir quelques nouvelles valeurs avec chaque version majeure (2.0 ou 1.6, par exemple), voire pour les versions intermédiaires (1.5r1 ou 1.5r2). Assurez-vous de tester votre application sur ces nouvelles cibles à chaque fois que cela est possible car certains peuvent utiliser ces nouvelles versions de terminaux dès qu'elles sortent.
- Le fait de tester avec des AVD, quelle que soit la cible, ne peut pas se substituer aux tests sur du vrai matériel. Les AVD sont conçus pour vous fournir des "environnements jetables" permettant de tester un grand nombre d'environnements, même ceux qui n'existent pas encore réellement. Cependant, vous devez mettre votre application à l'épreuve d'au moins un terminal Android. En outre, la vitesse de votre émulateur peut ne pas correspondre à celle du terminal – selon votre système, elle peut être plus rapide ou plus lente.

3

Création d'un squelette d'application

Tous les livres consacrés à un langage ou à un environnement de programmation commencent par présenter un programme de démonstration de type "Bonjour à tous !" : il permet de montrer que l'on peut construire quelque chose tout en restant suffisamment concis pour ne pas noyer le lecteur dans les détails. Cependant, le programme "Bonjour à tous !" classique n'est absolument pas interactif (il se contente d'écrire ces mots sur la console) et est donc assez peu stimulant.

Ce chapitre présentera donc un projet qui utilisera malgré tout un bouton et l'heure courante pour montrer le fonctionnement d'une activité Android simple.

Terminaux virtuels et cibles

Pour construire les projets, nous supposons que vous avez téléchargé le SDK (et, éventuellement, le plugin ADT d'Eclipse). Avant de commencer, nous devons présenter la notion

de "cible" car elle risque de vous surprendre et elle est relativement importante dans le processus de développement.

Comme son nom l'indique, un AVD (*Android Virtual Device*), est un terminal virtuel – par opposition aux vrais terminaux Android comme le G1 ou le Magic de HTC. Les AVD sont utilisés par l'émulateur fourni avec le SDK et vous permettent de tester vos programmes avant de les déployer sur les véritables terminaux. Vous devez indiquer à l'émulateur un terminal virtuel afin qu'il puisse prétendre qu'il est bien le terminal décrit par cet AVD.

Commencer par le début

Avec Android, comme nous l'avons vu au chapitre précédent, tout commence par la création d'un projet.

Si vous vous servez d'outils non Android, vous pouvez utiliser le script `android`, qui se trouve dans le répertoire `tools/` de l'installation du SDK en lui indiquant que vous souhaitez créer un projet (`create project`). Il faut alors lui indiquer la version de la cible, le répertoire racine du projet, le nom de l'activité et celui du paquetage Java où tout devra se trouver :

```
android create project \
       --target 2 \
       --path chemin/vers/mon/projet \
       --activity Now \
       --package com.commonsware.android.skeleton
```

Vous pouvez également télécharger les répertoires des projets exemples de ce livre sous la forme de fichiers ZIP à partir du site web de Pearson[1]. Ces projets sont prêts à être utilisés : vous n'avez donc pas besoin d'utiliser `android create project` lorsque vous aurez décompressé ces exemples.

Le répertoire `src/` de votre projet contient une arborescence de répertoires Java classique, créée d'après le paquetage Java que vous avez utilisé pour créer le projet (`com.commonsware.android` produit donc `src/com/commonsware/android/`). Dans le répertoire le plus bas, vous trouverez un fichier source nommé `Now.java`, dans lequel sera stocké le code de votre première activité.

Celle-ci sera constituée d'un unique bouton qui affichera l'heure à laquelle on a appuyé dessus pour le dernière fois (ou l'heure de lancement de l'application si on n'a pas encore appuyé sur ce bouton).

Si vous avez téléchargé les fichiers à partir du site web de Pearson, vous pouvez vous contenter d'utiliser directement le projet `Skeleton/Now`.

1. **http://www.pearson.fr**.

Ouvrez `Now.java` dans un éditeur de texte et copiez-y le code suivant :

```
package com.commonsware.android.skeleton;
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import java.util.Date;

public class Now extends Activity implements View.OnClickListener {
  Button btn;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    btn = new Button(this);
    btn.setOnClickListener(this);
    updateTime();
    setContentView(btn);
  }

  public void onClick(View view) {
    updateTime();
  }

  private void updateTime() {
    btn.setText(new Date().toString());
  }
}
```

Examinons maintenant chacune des parties de ce code.

Dissection de l'activité

La déclaration de paquetage doit être identique à celle que vous avez utilisée pour créer le projet.

Comme pour tout projet Java, vous devez importer les classes auxquelles vous faites référence. La plupart de celles qui sont spécifiques à Android se trouvent dans le paquetage `android` :

```
package com.commonsware.android.skeleton;

import android.app.Activity;
```

```
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import java.util.Date;
```

Notez bien que toutes les classes de Java SE ne sont pas utilisables par les programmes Android. Consultez le guide de référence des classes Android[1] pour savoir celles qui sont disponibles et celles qui ne le sont pas.

Les activités sont des classes publiques héritées de la classe de base `android.app.Activity`. Ici, l'activité contient un bouton (`btn`) :

```
public class Now extends Activity implements View.OnClickListener {
  Button btn;
```

Un bouton, comme vous pouvez le constater d'après le nom du paquetage, est un widget Android. Les widgets sont des éléments d'interface graphique que vous pouvez utiliser dans une application.

Pour rester simple, nous voulons capturer tous les clics de bouton dans l'activité elle-même : c'est la raison pour laquelle la classe de notre activité implémente également `OnClickListener`.

```
@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);

  btn = new Button(this);
  btn.setOnClickListener(this);
  updateTime();
  setContentView(btn);
}
```

La méthode `onCreate()` est appelée lorsque l'activité est lancée. La première chose à faire est d'établir un chaînage vers la superclasse afin de réaliser l'initialisation de l'activité Android de base.

Nous créons ensuite l'instance du bouton (avec `new Button(this)`) et demandons d'envoyer tous les clics sur ce bouton à l'instance de l'activité (avec `setOnClickListener()`). Nous appelons ensuite la méthode privée `updateTime()` (qui sera présentée plus loin), puis

1. **http://code.google.com/android/reference/packages.html**.

nous configurons la vue du contenu de l'activité pour que ce soit le bouton lui-même (avec `setContentView()`).

Tous les widgets dérivent de la classe de base `View`*. Bien que l'on construise généralement l'interface graphique à partir d'une hiérarchie de vues, nous n'utiliserons ici qu'une seule vue.*

Nous présenterons ce `Bundle icicle` magique au Chapitre 16. Pour l'instant, considérons-le comme un gestionnaire opaque, que toutes les activités reçoivent lors de leur création.

```
public void onClick(View view) {
  updateTime();
}
```

Avec Swing, un clic sur un `JButton` déclenche un `ActionEvent` qui est transmis à l'`ActionListener` configuré pour ce bouton. Avec Android, en revanche, un clic sur un bouton provoque l'appel de la méthode `onClick()` sur l'instance `OnClickListener` configurée pour le bouton. L'écouteur reçoit la vue qui a déclenché le clic (ici, il s'agit du bouton). Dans notre cas, nous nous contentons d'appeler la méthode privée `updateTime()` :

```
private void updateTime() {
  btn.setText(new Date().toString());
}
```

L'ouverture de l'activité (`onCreate()`) ou un clic sur le bouton (`onClick()`) doit provoquer la mise à jour du label du bouton avec la date courante. On utilise pour cela la méthode `setText()`, qui fonctionne exactement comme avec les `JButton` de Swing.

Compiler et lancer l'activité

Pour compiler l'activité, utilisez l'outil de création de paquetage Android intégré à votre IDE ou lancez `ant debug` depuis le répertoire racine de votre projet. Puis lancez l'activité en suivant les étapes suivantes :

1. Lancez l'émulateur en exécutant la commande `android`, en choisissant un AVD dans le gestionnaire d'AVD puis en cliquant sur le bouton Start. Vous devriez pouvoir vous contenter des valeurs par défaut qui vous sont présentées dans la boîte de dialogue des options. La Figure 3.1 montre l'écran d'accueil d'Android.

La première fois que vous utilisez un AVD avec l'émulateur, le temps de lancement est beaucoup plus long que les fois suivantes.

Figure 3.1
L'écran d'accueil d'Android.

2. Installez le paquetage (avec `ant install`, par exemple).
3. Consultez la liste des applications installées sur l'émulateur et recherchez celle qui s'appelle `Now` (voir Figure 3.2).

Figure 3.2
Le lanceur d'applications d'Android.

4. Ouvrez cette application.

Vous devriez voir apparaître un écran d'activité comme celui de la Figure 3.3.

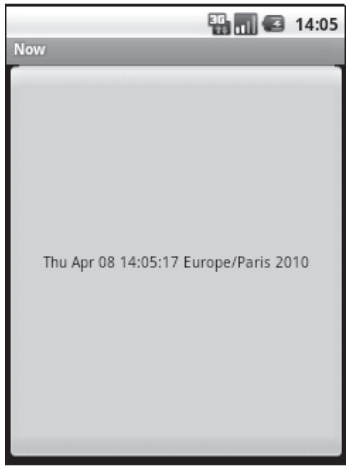


Figure 3.3
Démonstration de l'activité Now.

En cliquant sur le bouton – en d'autres termes, quasiment n'importe où sur l'écran du téléphone –, l'heure courante s'affichera sur le label du bouton.

Vous remarquerez que ce label est centré horizontalement et verticalement car c'est le style par défaut. Nous verrons au Chapitre 5 que nous pouvons évidemment contrôler ce formatage.

Une fois repu de cette technologie époustouflante des boutons, vous pouvez cliquer sur le bouton de retour en arrière de l'émulateur pour revenir au lanceur.

4

Utilisation des layouts XML

Bien qu'il soit techniquement possible de créer et d'attacher des composants widgets à une activité en utilisant uniquement du code Java comme nous l'avons fait au Chapitre 4, on préfère généralement employer un fichier de positionnement (*layout*) codé en XML. L'instanciation dynamique des widgets est réservée aux scénarios plus complexes, où les widgets ne sont pas connus au moment de la compilation (lorsqu'il faut, par exemple, remplir une colonne de boutons radio en fonction de données récupérées sur Internet).

Examinons maintenant ce code XML pour savoir comment sont agencées les vues des activités Android lorsque l'on utilise cette approche.

Qu'est-ce qu'un positionnement XML ?

Comme son nom l'indique, un positionnement XML est une spécification des relations existant entre les composants widgets – et avec leurs conteneurs (voir Chapitre 6) – exprimée sous la forme d'un document XML. Plus précisément, Android considère les layouts XML comme des ressources stockées dans le répertoire `res/layout` du projet.

Chaque fichier XML contient une arborescence d'éléments précisant le layout des widgets et les conteneurs qui composent une `View`. Les attributs de ces éléments sont des propriétés qui décrivent l'aspect d'un widget ou le comportement d'un conteneur. Un élément `Button` avec un attribut `android:textStyle = "bold"`, par exemple, signifie que le texte apparaissant sur ce bouton sera en gras.

L'outil `aapt` du SDK d'Android utilise ces layouts ; il est appelé automatiquement par la chaîne de production du projet (que ce soit *via* Eclipse ou par le traitement du fichier `build.xml` de `Ant`). C'est `aapt` qui produit le fichier source `R.java` du projet, qui vous permet d'accéder directement aux layouts et à leurs composants widgets depuis votre code Java, comme nous le verrons bientôt.

Pourquoi utiliser des layouts XML ?

La plupart de ce qui peut être fait avec des fichiers de positionnement XML peut également être réalisé avec du code Java. Vous pourriez, par exemple, utiliser `setTypeface()` pour qu'un bouton affiche son texte en gras au lieu d'utiliser une propriété dans un fichier XML. Ces fichiers s'ajoutant à tous ceux que vous devez déjà gérer, il faut donc qu'il y ait une bonne raison de les utiliser.

La principale est probablement le fait qu'ils permettent de créer des outils de définition des vues : le constructeur d'interfaces graphiques d'un IDE comme Eclipse ou un assistant dédié à la création des interfaces graphiques d'Android, comme DroidDraw[1], par exemple. Ces logiciels pourraient, en principe, produire du code Java au lieu d'un document XML, mais il est bien plus simple de relire la définition d'une interface graphique afin de la modifier lorsque cette définition est exprimée dans un format structuré comme XML au lieu d'être codée dans un langage de programmation. En outre, séparer ces définitions XML du code Java réduit les risques qu'une modification du code source perturbe l'application. XML est un bon compromis entre les besoins des concepteurs d'outils et ceux des programmeurs.

En outre, l'utilisation de XML pour la définition des interfaces graphiques est devenue monnaie courante. XAML (*eXtensible Application Markup Language*)[2] de Microsoft, Flex[3] d'Adobe et XUL (*XML User interface Language*)[4] de Mozilla utilisent toutes une approche équivalente de celle d'Android : placer les détails des positionnements dans un fichier XML en permettant de les manipuler à partir des codes sources (à l'aide de JavaScript pour XUL, par exemple). De nombreux frameworks graphiques moins connus, comme ZK[5], utilisent

1. **http://droiddraw.org/**.
2. **http://windowssdk.msdn.microsoft.com/en-us/library/ms752059.aspx**.
3. **http://www.adobe.com/products/flex/**.
4. **http://www.mozilla.org/projects/xul/**.
5. **http://www.zkoss.org/**.

également XML pour la définition de leurs vues. Bien que "suivre le troupeau" ne soit pas toujours le meilleur choix, cette politique a l'avantage de faciliter la transition vers Android à partir d'un autre langage de description des vues reposant sur XML.

Contenu d'un fichier layout

Voici le bouton de l'application du chapitre précédent, converti en un fichier XML que vous trouverez dans le répertoire `Layouts/NowRedux` des codes de ce livre :

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

Ici, le nom de l'élément XML est celui de la classe du widget, `Button`. Ce dernier étant fourni par Android, il suffit d'utiliser simplement son nom de classe. Dans le cas d'un widget personnalisé, dérivé d'`android.view.View`, il faudrait utiliser un nom pleinement qualifié, contenant le nom du paquetage (`com.commonsware.android.MonWidget`, par exemple).

L'élément racine doit déclarer l'espace de noms XML d'Android :

```
xmlns:android="http://schemas.android.com/apk/res/android"
```

Tous les autres éléments sont des fils de la racine et héritent de cette déclaration d'espace de noms.

Comme l'on souhaite pouvoir faire référence à ce bouton à partir de notre code Java, il faut lui associer un identifiant avec l'attribut `android:id`. Nous expliquerons plus précisément ce mécanisme plus loin dans ce chapitre.

Les autres attributs sont les propriétés de cette instance de `Button` :

- `android:text` précise le texte qui sera affiché au départ sur le bouton (ici, il s'agit d'une chaîne vide).
- `android:layout_width` et `android:layout_height` précisent que la largeur et la hauteur du bouton rempliront le "parent", c'est-à-dire ici l'écran entier. Ces attributs seront présentés plus en détail au Chapitre 6.

Ce widget étant le seul contenu de notre activité, nous n'avons besoin que de cet élément. Les vues plus complexes nécessitent une arborescence d'éléments, afin de représenter les widgets et les conteneurs qui contrôlent leur positionnement. Dans la suite de ce livre, nous

utiliserons des positionnements XML à chaque fois que cela est nécessaire : vous trouverez donc des dizaines d'exemples plus complexes que vous pourrez utiliser comme point de départ pour vos propres projets.

Identifiants des widgets

De nombreux widgets et conteneurs ne peuvent apparaître que dans le fichier de positionnement et ne seront pas utilisés par votre code Java. Le plus souvent, un label statique (`TextView`), par exemple, n'a besoin d'être dans le fichier XML que pour indiquer l'emplacement où il doit apparaître dans l'interface. Ce type d'élément n'a donc pas besoin d'un attribut `android:id` pour lui donner un nom.

En revanche, tous les éléments dont vous avez besoin dans votre source Java doivent posséder cet attribut.

La convention consiste à utiliser le format `@+id/`*`nom_unique`* comme valeur d'identifiant, où *`nom_unique`* représente le nom local du widget, qui doit être unique. Dans l'exemple de la section précédente, l'identifiant du widget `Button` était `@+id/button`.

Android utilise également quelques valeurs `android:id` spécifiques, de la forme `@android:id/...`. Nous les rencontrerons dans différents chapitres de ce livre.

Utilisation des widgets dans le code Java

Une fois que vous avez douloureusement configuré les widgets et les conteneurs dans un fichier de positionnement XML nommé `main.xml` et que vous l'avez placé dans le répertoire `res/layout`, vous n'avez plus qu'à ajouter une seule instruction dans la méthode `onCreate()` de votre activité pour pouvoir utiliser ce positionnement :

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

Il s'agit du même appel `setContentView()` que nous avions utilisé précédemment en lui passant une instance d'une sous-classe de `View` (un `Button`). La vue construite à partir de notre fichier XML est désormais accessible à partir de la classe `R`. Tous les positionnements définis se trouvent sous `R.layout`, indiqués par le nom de base du fichier – `R.layout.main` désigne donc `main.xml`.

Pour accéder à nos widgets, nous utilisons ensuite la méthode `findViewById()`, en lui passant l'identifiant numérique du widget concerné. Cet identifiant a été produit par Android dans la classe `R` et est de la forme `R.id.`*`qquechose`* (où *`qquechose`* est le widget que vous recherchez). Ces widgets sont des sous-classes de `View`, exactement comme l'instance de `Button` que nous avions créée au chapitre précédent.

Fin de l'histoire

Dans le premier exemple `Now`, le texte du bouton affichait l'heure à laquelle on avait appuyé dessus pour la dernière fois (ou l'heure à laquelle l'activité avait été lancée).

L'essentiel du code fonctionne encore, même dans cette nouvelle version (que nous appellerons `NowRedux`). Cependant, au lieu d'instancier le bouton dans la méthode `onCreate()`, nous faisons référence à celui qui est décrit dans l'agencement XML :

```
package com.commonsware.android.layouts;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import java.util.Date;

public class NowRedux extends Activity
  implements View.OnClickListener {
  Button btn;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    setContentView(R.layout.main);
    btn=(Button)findViewById(R.id.button);
    btn.setOnClickListener(this);
    updateTime();
  }
  public void onClick(View view) {
    updateTime();
  }
  private void updateTime() {
    btn.setText(new Date().toString());
  }
}
```

La première différence est qu'au lieu de créer la vue dans le code Java nous faisons référence au fichier XML de positionnement (avec `setContentView(R.layout.main)`). Le fichier source `R.java` sera mis à jour lorsque le projet sera recompilé, afin d'inclure une référence au fichier de positionnement (`main.xml` du répertoire `res/layout`).

La seconde différence est qu'il faut retrouver l'instance de notre bouton en appelant la méthode `findViewById()`. Comme l'identifiant de ce bouton est `@+id/button`, nous pouvons désigner son identifiant numérique par `R.id.button`. Il reste ensuite à mettre en place l'écouteur d'événement et à configurer son label.

La Figure 4.1 montre que le résultat est le même que celui de l'exemple `Now` précédent.

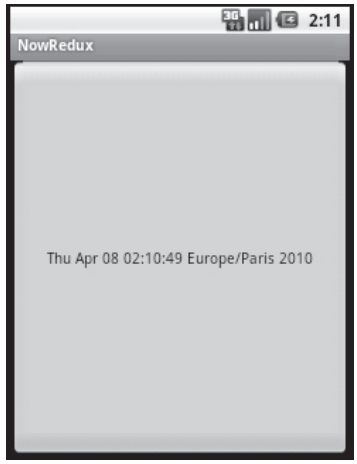


Figure 4.1
L'activité `NowRedux`.

5

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 20.

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

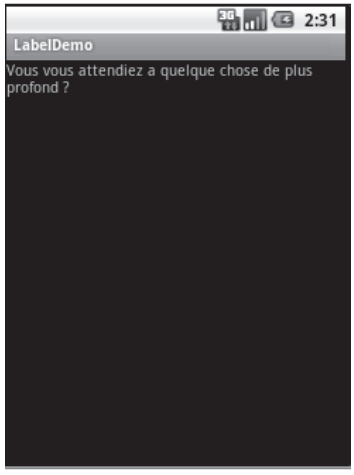


Figure 5.1
L'application `LabelDemo`.

Boutons

Nous avons déjà utilisé le widget `Button` dans les deux chapitres précédents. `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.

Cependant, Android 1.6 ajoute une nouvelle fonctionnalité permettant de déclarer un écouteur "de clic" (Listener) pour un bouton. Outre l'approche classique consistant à définir un objet (comme l'activité) comme une implémentation de l'interface `View.OnClickListener`, nous pouvons désormais utiliser une approche un peu plus simple :

- définir une méthode publique de l'activité qui prend un unique paramètre `View` et qui renvoie `void` ;
- dans l'élément `Button` du fichier de positionnement XML, inclure l'attribut `android:onClick` en précisant le nom de cette méthode.

Nous pourrions, par exemple, avoir une méthode de l'activité qui ressemble à celle-ci :

```
public void uneMethode(View leBouton) {
  // Faire quelque chose
}
```

On ajouterait ensuite l'attribut `android:onClick` à la déclaration XML concernant le bouton :

```
<Button
  android:onClick="uneMethode"
  ...
/>
```

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 Chapitre 20). 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 5.2.

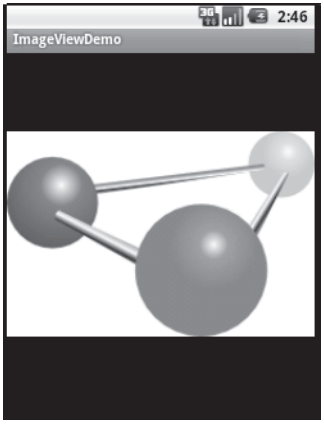


Figure 5.2
L'application `ImageViewDemo`.

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 ?).

La plupart de ces propriétés sont également disponibles avec le nouvel attribut `android:inputType` introduit par Android 1.5 pour lui ajouter des "claviers logiciels" (présentés au Chapitre 10).

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"
  />
```

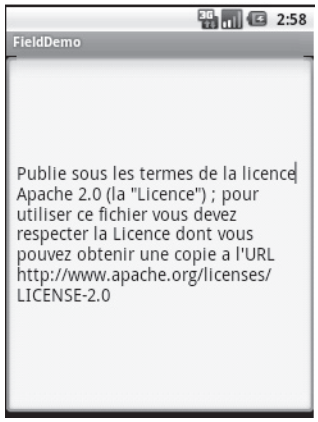
Vous remarquerez que la valeur d'`android:singleLine` est `false`, ce qui signifie que les utilisateurs pourront saisir plusieurs lignes de texte dans ce champ.

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 5.3 montre le résultat obtenu.

Figure 5.3
L'application `FieldDemo`.



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 9.

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`, par exemple) permettent également de formater ce widget.

Dans le 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 5.4 et 5.5.

Figure 5.4
L'application `CheckBoxDemo` *avec la case décochée.*

Figure 5.5
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 ils 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 5.6 montre le résultat obtenu en utilisant le projet Java de base, fourni par Android.

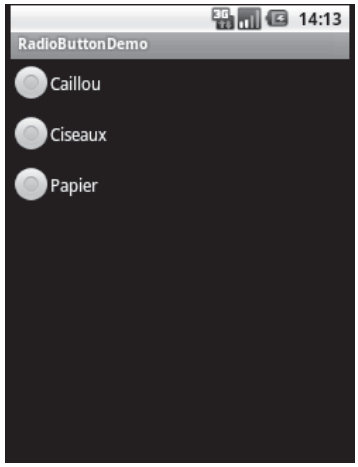


Figure 5.6
L'application `RadioButtonDemo`.

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 celles qui contrôlent la séquence de focus :

- `android:nextFocusDown` ;
- `android:nextFocusLeft` ;
- `android:nextFocusRight` ;
- `android:nextFocusUp`.

La propriété `android:visibility`, quant à elle contrôle la visibilité initiale du widget.

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()).

Couleurs

Les widgets Android disposent de deux types d'attributs pour contrôler la couleur. Certains, comme `android:background`, prennent une seule couleur (ou une image servant de fond). D'autres, comme `android:textColor`, s'appliquent à `TextView` (ou ses sous-classes) et peuvent prendre une `ColorStateList`.

Une `ColorStateList` permet de préciser plusieurs couleurs correspondant à différentes situations. Un `TextView` peut, par exemple, utiliser une couleur de texte particulière lorsqu'un élément est sélectionné dans une liste et une autre lorsqu'il ne l'est pas (nous présenterons les widgets de sélection au Chapitre 7). Tout ceci est géré par la `ColorStateList` associée au `TextView`.

Pour modifier la couleur d'un widget `TextView` à partir du code Java, deux choix sont possibles :

- Utiliser `ColorStateList.valueOf()`, qui renvoie une `ColorStateList` dans laquelle tous les états sont associés à la couleur fournie en paramètre à `valueOf()`. C'est l'équivalent Java de l'approche `android:textColor`, qui donne toujours la même couleur au `TextView` quelles que soient les circonstances.
- Créer une `ColorStateList` avec des valeurs différentes pour chaque état, soit au moyen du constructeur, soit *via* un document XML.

6

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 quatre conteneurs parmi les plus courants : `LinearLayout` (le modèle des boîtes), `RelativeLayout` (un modèle de positionnement relatif), `TableLayout` (le modèle en grille) et `ScrollView`, un conteneur conçu pour faciliter la mise en place des conteneurs avec barres de défilement.

Penser de façon linéaire : *LinearLayout*

`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()` du conteneur 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 le placement des autres widgets.

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. Le poids d'un widget est fixé à zéro par défaut.

Un autre moyen d'utiliser les poids consiste à allouer des pourcentages. Pour utiliser cette technique avec une disposition en ligne, par exemple :

- Initialisez à zéro les valeurs `android:layout_width` de tous les widgets du layout.
- Initialisez avec les pourcentages adéquats les valeurs `android:layout_weight` de tous les widgets du layout.
- Assurez-vous que la somme de ces pourcentages soit égale à 100.

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 sa méthode `setGravity()` en cours d'exécution) 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 en cours d'exécution la méthode `setPadding()` du widget).

La valeur de remplissage précise l'espace situé entre le contour de la "cellule" du widget et son contenu réel, comme le montre la Figure 6.1.

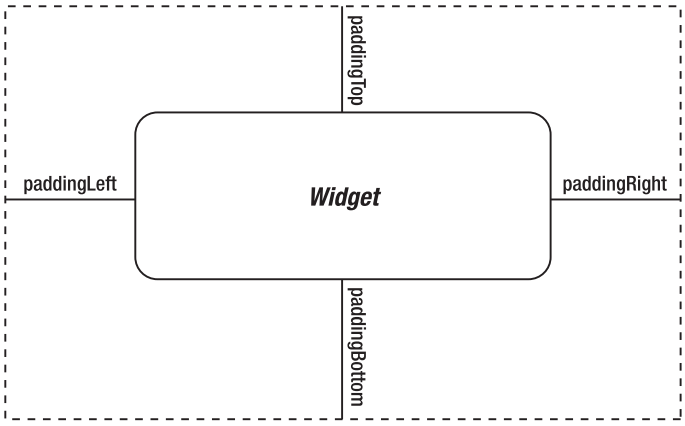


Figure 6.1
Relations entre un widget, sa cellule et ses valeurs de remplissage.

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`.

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

Si vous appliquez un fond au widget (avec l'attribut `android:background`, par exemple), ce fond sera placé à la fois derrière le widget et sa zone de remplissage. Si ce n'est pas ce que vous souhaitez, utilisez des marges plutôt que le remplissage car celles-ci ajoutent de l'espace sans augmenter la taille intrinsèque du widget. Pour configurer ces marges, utilisez les attributs comme `android:layout_marginTop`.

Exemple

Prenons un exemple d'application qui configure les propriétés d'un conteneur `LinearLayout` à la fois dans le fichier de description XML et en cours d'exécution. Voici le contenu du fichier `layout/main.xml` du projet `Containers/Linear` :

```
<?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 6.2 montre ce qu'affiche l'application lorsqu'elle est lancée dans l'émulateur.

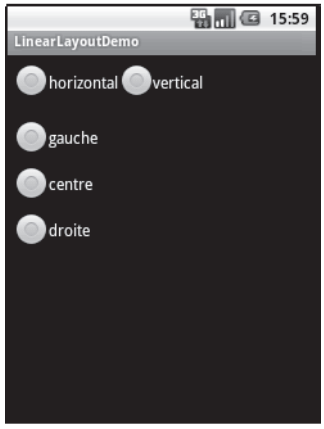
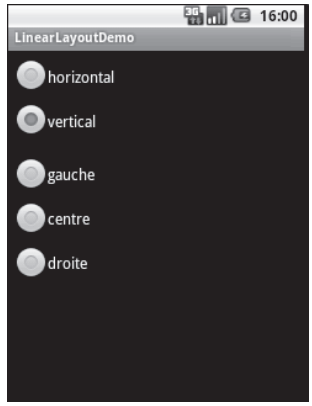


Figure 6.2
L'application `LinearLayoutDemo` *lors de son lancement.*

Si l'on clique sur le bouton `vertical`, le `RadioGroup` du haut s'ajuste en conséquence (voir Figure 6.3).

Figure 6.3
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 6.4 et 6.5).

Figure 6.4
La même application, avec les boutons `vertical` *et* `centre` *cochés.*

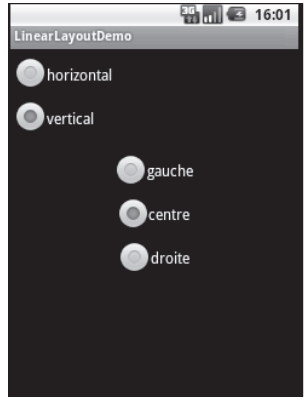
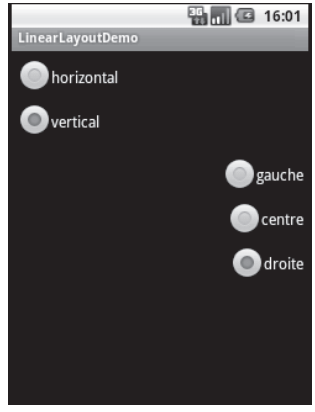


Figure 6.5
La même application, avec les boutons `vertical` *et* `droite` *cochés.*



Tout est relatif : *RelativeLayout*

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

Pour utiliser un conteneur `RelativeLayout`, 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`.

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

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 :

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

Avant la version 1.6, Android ne lisait qu'une seule fois le fichier XML et calculait donc en séquence la taille et la position de chaque widget. Ceci signifiait que l'on ne pouvait pas faire référence à un widget qui n'avait pas été défini plus haut dans le fichier, ce qui compliquait un peu les choses.

Désormais, Android traite les règles en effectuant deux passes : vous pouvez donc utiliser des références vers des widgets qui sont définis plus loin dans le fichier.

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.

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"`).

Bien sûr, ce remplissage de 15 pixels est un peu du bricolage. À partir d'Android 1.6, une meilleure solution consiste à ancrer le widget `EditText` en haut de l'écran et à aligner le `TextView` avec la ligne de base de celui-ci, comme le montre l'exemple qui suit (ce n'était pas possible dans les versions antérieures à cause de l'interprétation en une seule passe que nous avons évoquée plus haut).

```
<?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:layout_alignBaseline="@+id/entry"
    android:layout_alignParentLeft="true"/>
  <EditText
    android:id="@id/entry"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_toRightOf="@id/label"
    android:layout_alignParentTop="true"/>
  <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>
```

Sans rien changer au code Java qui a été produit automatiquement, l'émulateur affiche le résultat présenté à la Figure 6.6.

Figure 6.6
L'application `RelativeLayoutDemo`.

Tabula Rasa : *TableLayout*

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. Avec ce fragment XML, le champ de saisie `EditText` s'étendra sur trois colonnes.

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

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). Ceci 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 6.7.

Figure 6.7
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 6.8 montre ce qu'affichera l'émulateur d'Android au lancement de l'activité.

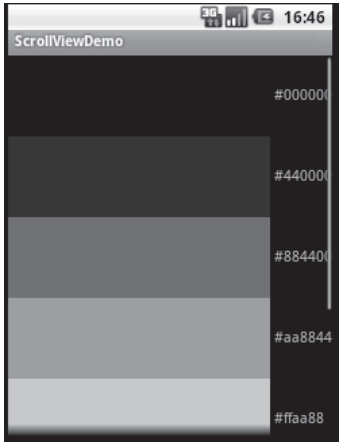


Figure 6.8
L'application `ScrollViewDemo`.

Vous remarquerez que l'on ne voit que cinq lignes. En pressant les boutons haut et 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.

Android 1.5 a ajouté `HorizontalScrollView`, qui fonctionne comme `ScrollView`, mais en horizontal. Ce défilement peut être utile pour les formulaires qui sont trop larges pour tenir en entier à l'écran. Notez que ni `HorizontalScrollView` ni `ScrollView` ne vous permettent de défiler dans les deux directions : vous devez choisir entre vertical ou horizontal.

7

Widgets de sélection

Au Chapitre 5, nous avons vu que les champs pouvaient imposer des contraintes sur leur contenu possible, afin de forcer une saisie uniquement numérique. Ce type de contrainte aide l'utilisateur à "faire ce qu'il faut" lorsqu'il entre des informations, notamment lorsqu'il s'agit d'un terminal mobile avec un clavier exigu.

La contrainte de saisie ultime consiste, évidemment, à ne proposer qu'une option possible parmi un ensemble de choix, ce qui peut être réalisé à l'aide des boutons radio que nous avons déjà présentés. Les kits de développement graphiques disposent également de listes déroulantes, et Android leur ajoute des widgets particulièrement adaptés aux dispositifs mobiles (`Gallery`, par exemple, permet d'examiner les photographies stockées sur le terminal).

En outre, Android permet de connaître aisément les choix qui sont proposés dans ces widgets. Plus précisément, il dispose d'un ensemble d'adaptateurs permettant de fournir une interface commune à toutes les listes de choix, que ce soient des tableaux statiques ou des bases de données. Les vues de sélection – les widgets pour présenter les listes de choix – sont transmises à un adaptateur pour fournir les choix possibles.

S'adapter aux circonstances

Dans l'absolu, les adaptateurs offrent une interface commune pour différentes API. Plus précisément, dans le cas d'Android, ils fournissent une interface commune au modèle de données sous-jacent d'un widget de sélection comme une liste déroulante. Cette utilisation des interfaces Java est assez classique (voir, par exemple, les adaptateurs de modèles pour `JTable` en Java/Swing), et Java n'est pas le seul environnement à fournir ce type d'abstraction (le framework XML de Flex accepte indifféremment du code XML en ligne ou téléchargé à partir d'Internet).

Les adaptateurs d'Android se chargent de fournir la liste des données d'un widget de sélection et de convertir les différents éléments en vues spécifiques pour qu'elles s'affichent dans ce widget de sélection. Ce dernier aspect des adaptateurs peut sembler un peu curieux mais, en réalité, il n'est pas si différent de ce que proposent les autres kits de développement graphiques pour redéfinir l'affichage par défaut. En Java/Swing, par exemple, si vous souhaitez qu'une liste implémentée par une `JList` soit, en réalité, une liste à cocher (où les différentes lignes sont composées d'une case à cocher et d'un label et où les clics modifient l'état de cette liste), vous finirez inévitablement par appeler la méthode `setCellRenderer()` pour disposer d'un objet `ListCellRenderer` qui, à son tour, permet de convertir le contenu d'une liste en widgets composites `JCheckBox`-plus-`JLabel`.

L'adaptateur le plus simple est `ArrayAdapter` puisqu'il suffit d'encapsuler un tableau ou une instance de `java.util.List` pour disposer d'un adaptateur prêt à fonctionner :

```
String[] items = {"ceci", "est", "une",
                  "liste", "vraiment", "stupide"};
new ArrayAdapter<String>(this,
  android.R.layout.simple_list_item_1, items);
```

Le constructeur d'`ArrayAdapter` attend trois paramètres :

- le contexte d'utilisation (généralement, il s'agit de l'instance de l'activité) ;
- l'identifiant de ressource de la vue à utiliser (un identifiant d'une ressource système prédéfinie, comme dans l'exemple précédent) ;
- le tableau ou la liste d'éléments à afficher.

Par défaut, `ArrayAdapter` appellera la méthode `toString()` des objets de la liste et enveloppera chaque chaîne ainsi obtenue dans la vue désignée par la ressource indiquée. `android.R.layout.simple_list_item_1` se contente de transformer ces chaînes en objets `TextView` qui, à leur tour, s'afficheront dans la liste, le spinner ou tout widget qui utilise cet `ArrayAdapter`. Au Chapitre 8, nous verrons comment créer une sous-classe d'`Array-`

`Adapter` pour redéfinir la création des lignes afin d'avoir un meilleur contrôle sur leur apparence.

Voici d'autres adaptateurs dont vous aurez certainement besoin :

- `CursorAdapter` convertit un `Cursor`, généralement fourni par un *content provider*, en un objet pouvant s'afficher dans une vue de sélection (nous étudierons plus en détail `CursorAdapter` au Chapitre 22, avec les bases de données).
- `SimpleAdapter` convertit les données trouvées dans les ressources XML.

Listes des bons et des méchants

Le widget classique d'Android pour les listes s'appelle `ListView`. Pour disposer d'une liste complètement fonctionnelle, il suffit d'inclure un objet `ListView` dans votre présentation, d'appeler `setAdapter()` pour fournir les données et les vues filles, puis d'attacher un écouteur *via* `setOnItemSelectedListener()` pour être prévenu de toute modification de la sélection.

Cependant, si votre activité est pilotée par une seule liste, il peut être préférable que cette activité soit une sous-classe de `ListActivity` plutôt que de la classe de base `Activity` traditionnelle. Si votre vue principale est uniquement constituée de la liste, vous n'avez même pas besoin de fournir de layout – `ListActivity` construira pour vous une liste qui occupera tout l'écran. Vous pouvez toutefois personnaliser cette présentation à condition d'identifier cette `ListView` par `@android:id/list`, afin que `ListActivity` sache quelle est la liste principale de l'activité.

Voici, par exemple, le fichier de disposition du projet `Selection/List`, qui affiche simplement une liste surmontée d'un label qui devra afficher en permanence la sélection courante :

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

```
  <ListView
    android:id="@android:id/list"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:drawSelectorOnTop="false"
    />
</LinearLayout>
```

Le code Java permettant de configurer cette liste et de la connecter au label est le suivant :

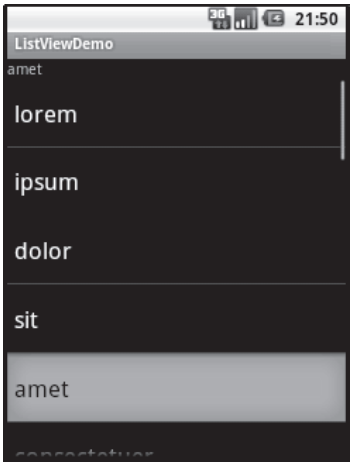
```
public class ListViewDemo extends ListActivity {
  TextView selection;
  String[] items={"lorem", "ipsum", "dolor", "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat",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

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    setListAdapter(new ArrayAdapter<String>(this,
                        android.R.layout.simple_list_item_1,
                        items));
    selection=(TextView)findViewById(R.id.selection);
  }

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

Vous pouvez configurer l'adaptateur d'une `ListActivity` par un appel à `setListAdapter()` – ici, on fournit un `ArrayAdapter` qui enveloppe un tableau de chaînes quelconques. Pour être prévenu des changements dans la liste de sélection, on redéfinit `onListItemClick()` pour qu'elle agisse de façon appropriée en tenant compte de la vue fille et de

la position qui lui sont passées en paramètre (ici, elle écrit dans le label le texte situé à cette position).

Le résultat est montré à la Figure 7.1.

Figure 7.1
L'application `ListViewDemo`.



Le second paramètre de notre `ArrayAdapter` – `android.R.layout.simple_list_item_1` – contrôle l'aspect des lignes. La valeur utilisée dans l'exemple précédent fournit une ligne Android standard : grande police, remplissage important et texte en blanc.

Par défaut, `ListView` est simplement configurée pour recevoir les clics sur les entrées de la liste. Cependant, on a parfois besoin qu'une liste mémorise un ou plusieurs choix de l'utilisateur ; `ListView` permet également de le faire, au prix de quelques modifications :

1. Dans le code Java, appelez la méthode `setChoiceMode()` de l'objet `ListView` afin de configurer le mode de sélection en lui passant en paramètre la constante `CHOICE_MODE_SINGLE` ou `CHOICE_MODE_MULTIPLE` (pour obtenir l'objet `ListView`, il suffit d'appeler la méthode `getListView()` à partir d'une `ListActivity`).
2. Puis, au lieu de passer en paramètre `android.R.layout.simple_list_item_1` au constructeur d'`ArrayAdapter`, utilisez soit `android.R.layout.simple_list_item_single_choice`, soit `android.R.layout.simple_list_item_multiple_choice` pour mettre en place, respectivement, une liste à choix unique (voir Figure 7.2) ou à choix multiples (voir Figure 7.3).
3. Pour savoir quel est l'élément choisi par l'utilisateur, appelez la méthode `getCheckedItemPositions()` du widget `ListView`.

Figure 7.2
Liste à choix unique.

Figure 7.3
Liste à choix multiples.

Contrôle du Spinner

Le `Spinner` d'Android est l'équivalent des boîtes déroulantes que l'on trouve dans certains kits de développement (`JComboBox` en Java/Swing, par exemple). Appuyer sur le bouton central du pad du terminal fait surgir une boîte de sélection permettant à l'utilisateur de

faire son choix. On peut ainsi faire son choix dans une liste sans occuper tout l'écran comme avec une `ListView`, mais au prix d'un clic supplémentaire ou d'un pointage sur l'écran.

Comme pour `ListView`, on fournit l'adaptateur pour les données et les vues filles *via* `setAdapter()` et on accroche un écouteur avec `setOnItemSelectedListener()`.

Si l'on souhaite personnaliser la vue d'affichage de la boîte déroulante, il faut configurer l'adaptateur, pas le widget `Spinner`. Pour ce faire, on a donc besoin de la méthode `setDropDownViewResource()` afin de fournir l'identifiant de la vue concernée.

Voici par exemple le fichier de disposition du projet `Selection/Spinner`, qui permet de mettre en place une vue simple contenant un `Spinner` :

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

Il s'agit de la même vue que celle de la section précédente, mais avec `Spinner` à la place de `ListView`. La propriété `android:drawSelectorOnTop` indique que la flèche permettant de dérouler la sélection se trouvera à droite du `Spinner`.

Voici le code Java permettant de remplir et d'utiliser le `Spinner` :

```
public class SpinnerDemo extends Activity
  implements AdapterView.OnItemSelectedListener {
  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);
      selection=(TextView)findViewById(R.id.selection);

      Spinner spin=(Spinner)findViewById(R.id.spinner);
      spin.setOnItemSelectedListener(this);

      ArrayAdapter<String> aa=new ArrayAdapter<String>(this,
                              android.R.layout.simple_spinner_item,
                              items);

      aa.setDropDownViewResource(
        android.R.layout.simple_spinner_dropdown_item);
      spin.setAdapter(aa);
    }

    public void onItemSelected(AdapterView<?> parent,
                               View v, int position, long id) {
      selection.setText(items[position]);
    }

    public void onNothingSelected(AdapterView<?> parent) {
      selection.setText("");
    }
  }
```

Ici, c'est l'activité elle-même qui sert d'écouteur de sélection (`spin.setOnItemSelectedListener(this)`), ce qui est possible car elle implémente l'interface `OnItemSelectedListener`. On configure l'adaptateur non seulement avec une liste de mots quelconques mais également avec une ressource spécifique qui servira à la vue déroulante (*via* `aa.setDropDownViewResource()`). Vous remarquerez également que l'on utilise la vue prédéfinie `android.R.layout.simple_spinner_item` pour afficher les éléments du `Spinner`. Enfin, on implémente les méthodes de rappels nécessaires d'`OnItemSelectedListener` pour que le contenu du label évolue en fonction du choix de l'utilisateur.

On obtient ainsi le résultat présenté aux Figures 7.4 et 7.5.

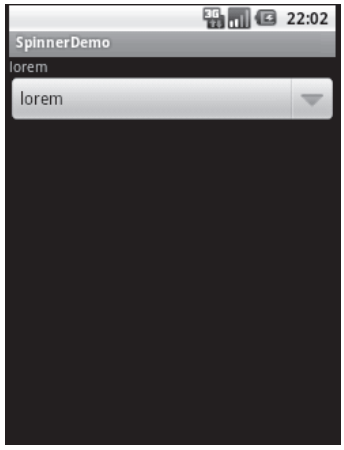


Figure 7.4
L'application `SpinnerDemo` *lors de son lancement.*

Figure 7.5
La même application avec affichage de la liste déroulante du `Spinner`.

Mettez vos lions en cage

Comme son nom l'indique, `GridView` vous offre une grille dans laquelle vous pouvez disposer vos choix. Vous avez un contrôle limité sur le nombre et la taille des colonnes ; le nombre de lignes est déterminé dynamiquement en fonction du nombre d'éléments rendus disponibles par l'adaptateur fourni.

Voici les propriétés qui, combinées, déterminent le nombre et la taille des colonnes :

- `android:numColumns` indique le nombre de colonnes ; si sa valeur est `auto_fit`, Android calculera ce nombre en fonction de l'espace disponible et de la valeur des autres propriétés.

- `android:verticalSpacing` et son homologue `android:horizontalSpacing` précisent l'espace séparant les éléments de la grille.
- `android:columnWidth` indique la largeur de chaque colonne en pixels.
- `android:stretchMode` indique, pour les grilles dont la valeur d'`android:numColumns` est `auto_fit`, ce qui devra se passer lorsqu'il reste de l'espace non occupé par des colonnes ou des espaces de séparation : si sa valeur est `columnWidth`, cet espace disponible sera pris par les colonnes ; si elle vaut `spacingWidth`, il sera absorbé par l'espacement entre les colonnes.

Supposons, par exemple, que l'écran fasse 320 pixels de large, que la valeur d'`android:columnWidth` soit de `100px` et celle d'`android:horizontalSpacing`, de `5px` : trois colonnes occuperaient donc 310 pixels (trois colonnes de 100 pixels et deux séparations de 5 pixels). Si `android:stretchMode` vaut `columnWidth`, les trois colonnes s'élargiront de 3-4 pixels pour utiliser les 10 pixels restants ; si `android:stretchMode` vaut `spacingWidth`, les deux espacements s'élargiront chacun de 5 pixels pour absorber ces 10 pixels.

Pour le reste, `GridView` fonctionne exactement comme n'importe quel autre widget de sélection – on utilise `setAdapter()` pour fournir les données et les vues filles, on appelle `setOnItemSelectedListener()` pour enregistrer un écouteur de choix, etc.

Voici, par exemple, le fichier de disposition XML du projet `Selection/Grid` :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <TextView
    android:id="@+id/selection"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    />
  <GridView
    android:id="@+id/grid"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:verticalSpacing="35px"
    android:horizontalSpacing="5px"
```

```
    android:numColumns="auto_fit"
    android:columnWidth="100px"
    android:stretchMode="columnWidth"
    android:gravity="center"
    />
</LinearLayout>
```

Cette grille occupe tout l'écran, sauf la partie réservée au label qui affiche la sélection courante. Le nombre de colonnes est calculé par Android (`android:numColumns = "auto_fit"`) à partir d'un espacement horizontal de 5 pixels (`android:horizontalSpacing = "5px"`) et d'une largeur de colonne de 100 pixels (`android:columnWidth = "100px"`). Les colonnes absorberont l'espace restant disponible (`android:stretchMode = "columnWidth"`).

Le code Java permettant de configurer cette grille est le suivant :

```
public class GridDemo extends Activity
  implements AdapterView.OnItemSelectedListener {
  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);
    selection=(TextView)findViewById(R.id.selection);

    GridView g=(GridView)findViewById(R.id.grid);
    g.setAdapter(new FunnyLookingAdapter(this,
                      android.R.layout.simple_list_item_1,
                      items));
    g.setOnItemSelectedListener(this);
  }

  public void onItemSelected(AdapterView<?> parent, View v,
                             int position, long id) {
    selection.setText(items[position]);
  }
```

```
    public void onNothingSelected(AdapterView<?> parent) {
      selection.setText("");
    }

    private class FunnyLookingAdapter extends ArrayAdapter {
      Context ctxt;

      FunnyLookingAdapter(Context ctxt, int resource,
                          String[] items) {
        super(ctxt, resource, items);

        this.ctxt=ctxt;
      }

      public View getView(int position, View convertView,
                          ViewGroup parent) {
        TextView label=(TextView)convertView;

        if (convertView==null) {
          convertView=new TextView(ctxt);
          label=(TextView)convertView;
        }

        label.setText(items[position]);

        return(convertView);
      }
    }
  }
```

Au lieu d'utiliser des widgets `TextView` automatiques pour les cellules de la grille, comme dans les sections précédentes, on crée nos propres vues qui héritent d'`ArrayAdapter` et redéfinissent `getView()`. Ici, on enveloppe les chaînes dans nos propres widgets `TextView`, juste pour changer un peu. Notre `getView()` se contente de réutiliser le texte du `TextView` qui lui est passé en paramètre ; si ce dernier vaut `null`, il en crée un et le remplit.

Avec l'espacement vertical de 35 pixels indiqué dans le fichier de description (`android:verticalSpacing = "35"`), la grille ne tiendra pas entièrement dans l'écran de l'émulateur (voir Figures 7.6 et 7.7).

Figure 7.6
L'application GridDemo lors de son démarrage.

Figure 7.7
La même application, après défilement vers le bas.

Champs : économisez 35 % de la frappe !

`AutoCompleteTextView` est une sorte d'hybride d'`EditText` (champ de saisie) et de `Spinner`.

Avec l'auto-complétion, le texte saisi par l'utilisateur est traité comme un préfixe de filtrage : il est comparé à une liste de préfixes candidats et les différentes correspondances s'affichent dans une liste de choix qui ressemble à un `Spinner`. L'utilisateur peut alors

continuer sa saisie (si le mot n'est pas dans la liste) ou choisir une entrée de celle-ci pour qu'elle devienne la valeur du champ.

`AutoCompleteTextView` étant une sous-classe d'`EditText`, vous pouvez utiliser toutes les propriétés de cette dernière pour contrôler son aspect – la police et la couleur du texte, notamment.

Vous pouvez fournir à `AutoCompleteTextView` un adaptateur contenant la liste des valeurs candidates à l'aide de `setAdapter()` mais, comme l'utilisateur peut très bien saisir un texte qui n'est pas dans cette liste, `AutoCompleteTextView` ne permet pas d'utiliser les écouteurs de sélection. Il est donc préférable d'enregistrer un `TextWatcher`, exactement comme n'importe quel `EditText`, pour être prévenu lorsque le texte a été modifié. Ce type d'événement est déclenché par une saisie manuelle ou par une sélection dans la liste des propositions.

Voici, par exemple, le fichier de description du projet `Selection/AutoComplete` :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <TextView
    android:id="@+id/selection"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    />
  <AutoCompleteTextView android:id="@+id/edit"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:completionThreshold="3"/>
</LinearLayout>
```

Le code Java correspondant est le suivant :

```
public class AutoCompleteDemo extends Activity
  implements TextWatcher {
  TextView selection;
  AutoCompleteTextView edit;
  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);
      selection=(TextView)findViewById(R.id.selection);
      edit=(AutoCompleteTextView)findViewById(R.id.edit);
      edit.addTextChangedListener(this);

      edit.setAdapter(new ArrayAdapter<String>(this,
                          android.R.layout.simple_dropdown_item_1line,
                          items));
    }

    public void onTextChanged(CharSequence s, int start, int before,
                              int count) {
      selection.setText(edit.getText());
    }

    public void beforeTextChanged(CharSequence s, int start,
                                  int count, int after) {
      // imposée par l'interface, mais inutilisée
    }

    public void afterTextChanged(Editable s) {
      // imposée par l'interface, mais inutilisée
    }
  }
```

Cette fois-ci, notre activité implémente l'interface `TextWatcher`, ce qui signifie que nos méthodes de rappel doivent se nommer `onTextChanged()` et `beforeTextChanged()`. Ici, seule la première nous intéresse : elle modifie le label de sélection pour qu'il reflète le choix courant du champ `AutoCompleteTextView`.

Les Figures 7.8, 7.9 et 7.10 montrent ce qu'affiche cette application.

Figure 7.8
L'application `AutoCompleteDemo` *après son démarrage.*

Figure 7.9
La même application après avoir saisi quelques lettres. La liste des propositions apparaît dans une liste déroulante.

Figure 7.10
La même application, après avoir choisi le texte suggéré.

Galeries

Le widget `Gallery` n'existe généralement pas dans les autres kits de développement graphiques. En réalité, il s'agit d'une liste horizontale, où chaque choix défile selon l'axe horizontal et où l'élément sélectionné est mis en surbrillance. Sur un terminal Android, l'utilisateur peut parcourir les différents choix à l'aide des boutons gauche et droit du pad.

`Gallery` prend moins de place à l'écran que `ListView`, tout en montrant plusieurs choix à la fois (pour autant qu'ils soient suffisamment courts). Par rapport à `Spinner`, `Gallery` montre également plusieurs choix simultanément.

L'exemple canonique d'utilisation de `Gallery` consiste à parcourir une galerie de photos – l'utilisateur peut ainsi prévisualiser les imagettes correspondant à une collection de photos ou d'icônes afin d'en choisir une.

Du point de vue du code, un objet `Gallery` fonctionne quasiment comme un `Spinner` ou un `GridView`. Il dispose de plusieurs propriétés :

- `android:spacing` indique le nombre de pixels séparant les différents éléments de la liste.
- `android:spinnerSelector` précise ce qui indiquera une sélection – il peut s'agir d'une référence à un objet `Drawable` (voir Chapitre 20) ou une valeur RGB de la forme `#AARRGGBB` ou équivalente.
- `android:drawSelectorOnTop` indique si la barre de sélection (ou le `Drawable`) doit être dessinée avant (`false`) ou après (`true`) le dessin du fils sélectionné. Si cette propriété vaut `true`, assurez-vous que le sélecteur soit suffisamment transparent pour que l'on puisse apercevoir le fils derrière lui ; sinon les utilisateurs ne pourront pas voir ce qu'ils ont choisi.

8

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 en lignes, 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]);
    }
  }
```

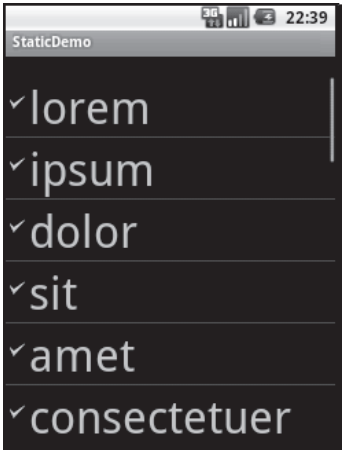
On peut remarquer que cette structure générale est identique à celle du projet `Selection/List` du Chapitre 7.

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 8.1, toutes les icônes sont les mêmes.

Figure 8.1
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.

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. Ceci 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".

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 nous appartient donc de la personnaliser et de la remplir comme nous le souhaitons 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 8.2).

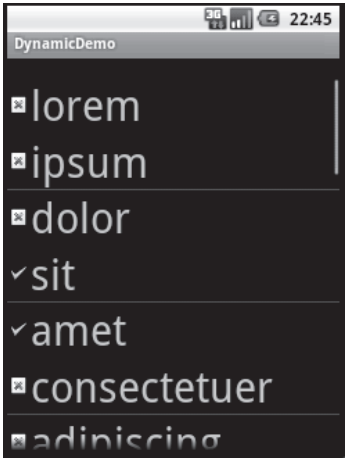


Figure 8.2
L'application `DynamicDemo`.

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.

Étudions quelques astuces permettant de rendre les widgets `ListView` plus efficaces.

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.

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".

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 ce 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.

Création d'une liste...

Les listes avec de belles icônes sont jolies, mais ne pourrions-nous pas créer des widgets `ListView` dont les lignes contiendraient des widgets fils interactifs et non plus passifs comme `TextView` et `ImageView` ? Pourrions-nous, par exemple, combiner une `RatingBar` avec du texte afin de permettre à l'utilisateur de faire défiler une liste de chansons et de les évaluer directement dans cette liste ?

Il y a une bonne et une mauvaise nouvelle.

La bonne est que l'on peut mettre des widgets interactifs dans les lignes ; la mauvaise est que c'est un peu compliqué, notamment lorsqu'il faut intervenir parce que l'état du widget interactif a changé (une valeur a été tapée dans un champ, par exemple). Il faut en effet stocker cet état quelque part puisque notre widget `RatingBar` sera recyclé lors du défilement de la `ListView`. On doit pouvoir configurer l'état de la `RatingBar` en fonction du mot qui est visible lorsque la `RatingBar` est recyclée et sauvegarder cet état, afin de le restaurer plus tard lorsque cette ligne précise redeviendra visible.

Par défaut, la `RatingBar` n'a absolument aucune idée de la manière dont les données de l'`ArrayAdapter` doivent être affichées. Après tout, une `RatingBar` n'est qu'un widget utilisé dans une ligne d'une `ListView`. Nous devons donc apprendre aux lignes quels sont les modèles qu'elles affichent, afin qu'elles sachent quel état de modèle modifier lorsque leurs barres d'évaluation sont cochées.

Étudions l'activité du projet `FancyLists/RateList`. On utilisera ici les mêmes classes de base que celles de l'exemple précédent – on affiche une liste de mots quelconques que l'on pourra évaluer. Les mots ayant la note maximale apparaîtront tout en majuscules.

```
public class RateListDemo 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);

  ArrayList<RowModel> list=new ArrayList<RowModel>();

  for (String s : items) {
    list.add(new RowModel(s));
  }

  setListAdapter(new CheckAdapter(this, list));
  selection=(TextView)findViewById(R.id.selection);
}

private RowModel getModel(int position) {
  return(((CheckAdapter)getListAdapter()).getItem(position));
}

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

class CheckAdapter extends ArrayAdapter<RowModel> {
  Activity context;

  CheckAdapter(Activity context, ArrayList<RowModel> list) {
    super(context, R.layout.row, list);

    this.context=context;
  }

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

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

      row=inflater.inflate(R.layout.row, null);
      wrapper=new ViewWrapper(row);
      row.setTag(wrapper);
      rate=wrapper.getRatingBar();

      RatingBar.OnRatingBarChangeListener l=
                  new RatingBar.OnRatingBarChangeListener() {
        public void onRatingChanged(RatingBar ratingBar,
                                    float rating,
                                    boolean fromTouch)  {
          Integer myPosition=(Integer)ratingBar.getTag();
          RowModel model=getModel(myPosition);

          model.rating=rating;

          LinearLayout parent=(LinearLayout)ratingBar.getParent();
          TextView label=(TextView)parent.findViewById(R.id.label);

          label.setText(model.toString());
        }
      };

      rate.setOnRatingBarChangeListener(l);
    }
    else {
      wrapper=(ViewWrapper)row.getTag();
      rate=wrapper.getRatingBar();
    }
    RowModel model=getModel(position);

    wrapper.getLabel().setText(model.toString());
    rate.setTag(new Integer(position));
    rate.setRating(model.rating);

    return(row);
  }
}

class RowModel {
```

```
    String label;
    float rating=2.0f;

    RowModel(String label) {
      this.label=label;
    }

    public String toString() {
      if (rating>=3.0) {
        return(label.toUpperCase());
      }

      return(label);
    }
  }
}
```

Les différences entre cette activité et la précédente sont les suivantes :

- Bien que nous utilisions toujours un tableau de `String` pour stocker la liste des mots, on le transforme en liste d'objets `RowModel` au lieu de le fournir à un `ArrayAdapter`. Ici, le `RowModel` est un ersatz de modèle mutable car il se contente de combiner un mot et son état de sélection. Dans un vrai système, il pourrait s'agir d'objets remplis à partir d'un `Cursor` et les propriétés auraient des significations métier plus importantes.
- Les méthodes utilitaires comme `onListItemClick()` doivent être modifiées pour refléter les différences entre un modèle `String` pur et l'utilisation d'un `RowModel`.
- Dans `getView()`, la sous-classe d'`ArrayAdapter` (`CheckAdapter`) teste si `convertView` est `null`, auquel cas elle crée une nouvelle ligne par inflation d'un layout simple (voir le code qui suit) et lui attache un `ViewWrapper`. Pour la `RatingBar` de la ligne, on ajoute un écouteur `onRatingChanged()` anonyme qui examine le marqueur de la ligne (`getTag()`) et le convertit en entier représentant la position dans l'`ArrayAdapter` de ce qu'affiche cette ligne. La barre d'évaluation peut alors obtenir le bon `RowModel` pour la ligne et mettre à jour le modèle en fonction de son nouvel état. Elle modifie également le texte situé à côté de la barre pour qu'il corresponde à l'état de celle-ci.
- On s'assure toujours que la `RatingBar` a le bon contenu et dispose d'un marqueur (*via* `setTag()`) pointant vers la position de la ligne dans l'adaptateur.

La disposition de la ligne est très simple : elle contient une `RatingBar` et un label `TextView` placés dans un conteneur `LinearLayout` :

```
<?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"
  >
    <RatingBar
      android:id="@+id/rate"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:numStars="3"
      android:stepSize="1"
      android:rating="2" />
    <TextView
      android:id="@+id/label"
      android:paddingLeft="2px"
      android:paddingRight="2px"
      android:paddingTop="2px"
      android:textSize="40sp"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"/>
  </LinearLayout>
```

Le `ViewWrapper` est tout aussi simple, car il se contente d'extraire ces deux widgets de la `View` de la ligne :

```
class ViewWrapper {
  View base;
  RatingBar rate=null;
  TextView label=null;

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

  RatingBar getRatingBar() {
    if (rate==null) {
      rate=(RatingBar)base.findViewById(R.id.rate);
    }

    return(rate);
  }

  TextView getLabel() {
```

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

        return(label);
      }
    }
```

Les Figures 8.3 et 8.4 montrent ce qu'affiche cette application.

Figure 8.3
L'application `RateListDemo` *lors de son démarrage.*

Figure 8.4
La même application, montrant un mot avec la note maximale.

… Et la vérifier deux fois

La liste d'évaluation de la section précédente fonctionne, mais son implémentation est très lourde. Pire, l'essentiel de ce code ennuyeux ne sera pas réutilisable, sauf dans des circonstances très limitées.

Nous pouvons mieux faire...

En fait, nous voudrions pouvoir créer un layout comme celui-ci :

```
<?xml version="1.0" encoding="utf-8"?>

  <com.commonsware.android.fancylists.seven.RateListView
    android:id="@android:id/list"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:drawSelectorOnTop="false"
  />
```

où toute la logique du code qui utilisait une `ListView` auparavant "fonctionnerait" avec la `RateListView` du layout :

```
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,
                     android.R.layout.simple_list_item_1,
                     items));
  selection=(TextView)findViewById(R.id.selection);
}

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

Les choses se compliquent un tout petit peu lorsqu'on réalise que, jusqu'à maintenant, les codes de ce chapitre n'ont jamais réellement modifié la `ListView` elle-même. Nous n'avons fait que travailler sur les adaptateurs, en redéfinissant `getView()`, en créant nos propres lignes par inflation, etc.

Si l'on souhaite que `RateListView` prenne n'importe quel `ListAdapter` et fonctionne "comme il faut", en plaçant les barres d'évaluation sur les lignes comme il se doit, nous devons faire un peu de gymnastique. Plus précisément, nous devons envelopper le `ListAdapter` "brut" dans un autre, qui sait comment placer les barres dans les lignes et mémoriser l'état de ces barres.

Nous devons d'abord établir le motif de conception dans lequel un `ListAdapter` en augmente un autre. Voici le code d'`AdapterWrapper`, qui prend en charge un `ListAdapter` et délègue toutes les méthodes de l'interface à `delegate`. Vous retrouverez ce code dans le projet `FancyLists/RateListView` :

```
public class AdapterWrapper implements ListAdapter {
  ListAdapter delegate=null;

  public AdapterWrapper(ListAdapter delegate) {
    this.delegate=delegate;
  }

  public int getCount() {
    return(delegate.getCount());
  }

  public Object getItem(int position) {
    return(delegate.getItem(position));
  }

  public long getItemId(int position) {
    return(delegate.getItemId(position));
  }

  public View getView(int position, View convertView,
                      ViewGroup parent) {
    return(delegate.getView(position, convertView, parent));
  }

  public void registerDataSetObserver(DataSetObserver observer) {
    delegate.registerDataSetObserver(observer);
  }
```

```
  public boolean hasStableIds() {
    return(delegate.hasStableIds());
  }

  public boolean isEmpty() {
    return(delegate.isEmpty());
  }

  public int getViewTypeCount() {
    return(delegate.getViewTypeCount());
  }

  public int getItemViewType(int position) {
    return(delegate.getItemViewType(position));
  }

  public void unregisterDataSetObserver(DataSetObserver observer) {
    delegate.unregisterDataSetObserver(observer);
  }

  public boolean areAllItemsEnabled() {
    return(delegate.areAllItemsEnabled());
  }

  public boolean isEnabled(int position) {
    return(delegate.isEnabled(position));
  }
}
```

Nous pouvons alors hériter d'`AdapterWrapper` pour créer `RateableWrapper`, qui redéfinit la méthode `getView()` tout en laissant le `ListAdapter` délégué faire "le vrai travail" :

```
public class RateableWrapper extends AdapterWrapper {
  Context ctxt=null;
  float[] rates=null;

  public RateableWrapper(Context ctxt, ListAdapter delegate) {
    super(delegate);

    this.ctxt=ctxt;
    this.rates=new float[delegate.getCount()];
```

```
    for (int i=0;i<delegate.getCount();i++) {
      this.rates[i]=2.0f;
    }
  }

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

    if (convertView==null) {
      LinearLayout layout=new LinearLayout(ctxt);
      RatingBar rate=new RatingBar(ctxt);

      rate.setNumStars(3);
      rate.setStepSize(1.0f);

      View guts=delegate.getView(position, null, parent);

      layout.setOrientation(LinearLayout.HORIZONTAL);

      rate.setLayoutParams(new LinearLayout.LayoutParams(
            LinearLayout.LayoutParams.WRAP_CONTENT,
            LinearLayout.LayoutParams.FILL_PARENT));
      guts.setLayoutParams(new LinearLayout.LayoutParams(
            LinearLayout.LayoutParams.FILL_PARENT,
            LinearLayout.LayoutParams.FILL_PARENT));

      RatingBar.OnRatingBarChangeListener l=
                    new RatingBar.OnRatingBarChangeListener() {
        public void onRatingChanged(RatingBar ratingBar,
                                    float rating,
                                    boolean fromTouch)  {
          rates[(Integer)ratingBar.getTag()]=rating;
        }
      };

      rate.setOnRatingBarChangeListener(l);

      layout.addView(rate);
      layout.addView(guts);

      wrap=new ViewWrapper(layout);
```

```
        wrap.setGuts(guts);
        layout.setTag(wrap);

        rate.setTag(new Integer(position));
        rate.setRating(rates[position]);

        row=layout;
      }
      else {
        wrap=(ViewWrapper)convertView.getTag();
        wrap.setGuts(delegate.getView(position, wrap.getGuts(),
                                      parent));
        wrap.getRatingBar().setTag(new Integer(position));
        wrap.getRatingBar().setRating(rates[position]);
      }

      return(row);
    }
  }
```

L'essentiel du traitement de notre liste d'évaluation réside dans `RateableWrapper`. Cette classe place les barres d'évaluation sur les lignes et mémorise leurs états lorsqu'elles sont modifiées par l'utilisateur. Pour stocker ces états, on utilise un tableau de `float` dont la taille correspond au nombre de lignes que `delegate` rapporte pour cette liste.

L'implémentation de la méthode `getView()` de `RateableWrapper` rappelle celle de `RateListDemo`, sauf qu'au lieu d'utiliser `LayoutInflater` on doit construire manuellement un conteneur `LinearLayout` pour y placer notre `RatingBar` et les "tripes" (c'est-à-dire toutes les vues créées par `delegate` et que l'on décore avec une barre d'évaluation). `LayoutInflater` est conçue pour construire une `View` à partir de widgets bruts or, ici, nous ne savons pas par avance à quoi ressembleront les lignes, hormis le fait qu'il faudra leur ajouter une barre d'évaluation. Le reste du code est semblable à celui de `RateListDemo` :

```
class ViewWrapper {
  ViewGroup base;
  View guts=null;
  RatingBar rate=null;

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

  RatingBar getRatingBar() {
    if (rate==null) {
```

```
      rate=(RatingBar)base.getChildAt(0);
    }

    return(rate);
  }

  void setRatingBar(RatingBar rate) {
    this.rate=rate;
  }

  View getGuts() {
    if (guts==null) {
      guts=base.getChildAt(1);
    }

    return(guts);
  }

  void setGuts(View guts) {
    this.guts=guts;
  }
}
```

Lorsque tout ceci est en place, le code de `RateListView` devient assez simple :

```
public class RateListView extends ListView {
  public RateListView(Context context) {
    super(context);
  }

  public RateListView(Context context, AttributeSet attrs) {
    super(context, attrs);
  }

  public RateListView(Context context, AttributeSet attrs,
                      int defStyle) {
    super(context, attrs, defStyle);
  }

  public void setAdapter(ListAdapter adapter) {
    super.setAdapter(new RateableWrapper(getContext(), adapter));
  }
}
```

On hérite simplement de `ListView` et on redéfinit `setAdapter()` pour pouvoir envelopper dans notre propre `RateableWrapper` le `ListAdapter` fourni en paramètre.

Comme le montre la Figure 8.5, les résultats sont identiques à ceux de `RateListDemo`, sauf que les mots ayant la note maximale n'apparaissent plus en majuscules.

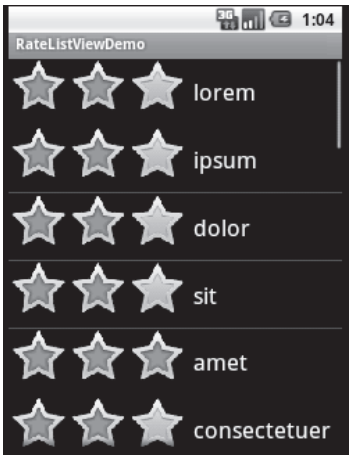


Figure 8.5
L'application `RateListViewDemo`.

La différence est la réutilisabilité. Nous pourrions assembler `RateListView` dans son propre fichier JAR et l'insérer dans n'importe quel projet Android qui en a besoin. Par conséquent, bien que `RateListView` soit un peu plus compliquée à écrire, il ne faut plus le faire qu'une seule fois et le reste du code de l'application est merveilleusement simple.

Cette classe `RateListView` *pourrait, bien sûr, proposer des fonctionnalités supplémentaires, comme la possibilité de modifier par programme l'état des barres (en mettant à jour le tableau de* `float` *et la* `RatingBar` *elle-même), autoriser l'appel d'un autre code lorsque l'état d'une barre est modifié (*via *une fonction de rappel), etc. Nous les laissons en exercice au lecteur.*

Adapter d'autres adaptateurs

Toutes les classes adaptateurs peuvent suivre le modèle de conception consistant à redéfinir `getView()` pour créer les lignes de la liste.

Cependant, `CursorAdapter` et ses sous-classes fournissent une implémentation par défaut de `getView()`, qui inspecte la `View` qui lui est passée en paramètre pour la recycler et appelle `newView()` si elle est `null`, ou `bindView()` dans le cas contraire.

Si vous étendez `CursorAdapter`, qui sert à afficher le résultat d'une requête adressée à une base de données ou à un fournisseur de contenu, il est donc préférable de redéfinir `newView()` et `bindView()` plutôt que `getView()`.

Pour cela, il suffit de supprimer le test `if` de `getView()` et de placer chaque branche de ce test dans deux méthodes séparées, comme celles-ci :

```
public View newView(Context context, Cursor cursor,
                    ViewGroup parent) {
  LayoutInflater inflater=context.getLayoutInflater();
  View row=inflater.inflate(R.layout.row, null);
  ViewWrapper wrapper=new ViewWrapper(row);
  row.setTag(wrapper);
  bindView(row, context, cursor);

  return(row);
}
public void bindView(View row, Context context, Cursor cursor) {
  ViewWrapper wrapper=(ViewWrapper)row.getTag();
  // Code pour remplir la ligne à partir du Cursor
}
```

L'utilisation de `Cursor` sera décrite au Chapitre 22.

9

Utiliser de jolis widgets et de beaux conteneurs

Les widgets et les conteneurs que nous avons présentés jusqu'à maintenant ne se trouvent pas seulement dans la plupart des kits de développement graphiques (sous une forme ou sous une autre), mais sont également très utilisés dans le développement des applications graphiques, qu'il s'agisse d'applications web, pour les PC ou pour les téléphones. Nous allons maintenant nous intéresser à des widgets et à des conteneurs un peu moins fréquents, mais néanmoins très utiles.

Choisir

Avec des terminaux ayant des capacités de saisie limitées comme les téléphones, il est très utile de disposer de widgets et de boîtes de dialogue capables d'anticiper ce que l'utilisateur veut taper. Cela minimise le nombre de frappes au clavier et de touches à l'écran et réduit les risques d'erreur (la saisie d'une lettre à la place d'un chiffre, par exemple).

Comme on l'a mentionné au Chapitre 5, `EditText` possède des variantes permettant de saisir des nombres ou du texte. Android dispose également de widgets (`DatePicker`, `TimePicker`) et de dialogues (`DatePickerDialog`, `TimePickerDialog`) facilitant la saisie des dates et des heures.

`DatePicker` et `DatePickerDialog` permettent de fixer une date de départ, sous la forme d'une année, d'un mois et d'un jour. Les mois vont de 0 (janvier) à 11 (décembre). Vous pouvez également préciser un écouteur (`OnDateChangedListener` ou `OnDateSetListener`) qui sera informé lorsqu'une nouvelle date a été choisie. Il vous appartient de stocker cette date quelque part, notamment si vous utilisez la boîte de dialogue, car vous n'aurez pas d'autre moyen d'obtenir ensuite la date choisie.

De même, `TimePicker` et `TimePickerDialog` permettent de fixer l'heure initiale, que l'utilisateur peut ensuite ajuster sous la forme d'une heure (de 0 à 23) et de minutes (de 0 à 59). Vous pouvez indiquer si le format de la date choisi utilise le mode sur 12 heures avec un indicateur AM/PM ou le mode 24 heures (ce qui, aux États-Unis, est appelé "temps militaire" et qui est le mode utilisé partout ailleurs dans le monde). Vous pouvez également fournir un écouteur (`OnTimeChangedListener` ou `OnTimeSetListener`) pour être prévenu du choix d'une nouvelle heure, qui vous sera fournie sous la forme d'une heure et de minutes.

Le projet `Fancy/Chrono`, par exemple, utilise une disposition très simple, formée d'un label et de deux boutons qui feront surgir les boîtes de dialogue pour choisir une date et une heure :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <TextView android:id="@+id/dateAndTime"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    />
  <Button android:id="@+id/dateBtn"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Choisir une date"
    />
  <Button android:id="@+id/timeBtn"
```

```
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Choisir une heure"
    />
</LinearLayout>
```

La partie intéressante se trouve dans le code Java :

```
public class ChronoDemo extends Activity {
  DateFormat fmtDateAndTime=DateFormat.getDateTimeInstance();
  TextView dateAndTimeLabel;
  Calendar dateAndTime=Calendar.getInstance();
  DatePickerDialog.OnDateSetListener d=new DatePickerDialog.OnDateSetListener()
{
    public void onDateSet(DatePicker view, int year, int monthOfYear,
                int dayOfMonth) {
      dateAndTime.set(Calendar.YEAR, year);
      dateAndTime.set(Calendar.MONTH, monthOfYear);
      dateAndTime.set(Calendar.DAY_OF_MONTH, dayOfMonth);
      updateLabel();
    }
  };
  TimePickerDialog.OnTimeSetListener t=new TimePickerDialog.OnTimeSetListener()
{
    public void onTimeSet(TimePicker view, int hourOfDay,
                          int minute) {
      dateAndTime.set(Calendar.HOUR_OF_DAY, hourOfDay);
      dateAndTime.set(Calendar.MINUTE, minute);
      updateLabel();
    }
  };

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

    Button btn=(Button)findViewById(R.id.dateBtn);

    btn.setOnClickListener(new View.OnClickListener() {
      public void onClick(View v) {
        new DatePickerDialog(ChronoDemo.this,
            d,
            dateAndTime.get(Calendar.YEAR),
```

```
                dateAndTime.get(Calendar.MONTH),
                dateAndTime.get(Calendar.DAY_OF_MONTH)).show();
            }
        });

        btn=(Button)findViewById(R.id.timeBtn);

        btn.setOnClickListener(new View.OnClickListener() {
          public void onClick(View v) {
            new TimePickerDialog(ChronoDemo.this,
                        t,
                        dateAndTime.get(Calendar.HOUR_OF_DAY),
                        dateAndTime.get(Calendar.MINUTE),
                        true).show();
          }
        });

        dateAndTimeLabel=(TextView)findViewById(R.id.dateAndTime);

        updateLabel();
      }

      private void updateLabel() {
        dateAndTimeLabel.setText(fmtDateAndTime
                                 .format(dateAndTime.getTime()));
      }
    }
```

Le "modèle" de cette activité est simplement une instance de `Calendar` initialisée avec la date et l'heure courantes et placée dans la vue *via* un formateur `DateFormat`. La méthode `updateLabel()` prend le `Calendar` courant, le formate et le place dans le `TextView` correspondant au label.

Chaque bouton est associé à un écouteur `OnClickListener` qui se charge d'afficher une boîte de dialogue `DatePickerDialog` ou `TimePickerDialog` selon le bouton sur lequel on clique. Dans le cas de `DatePickerDialog`, on passe un écouteur `OnDateSetListener` à `DatePickerDialog` pour mettre à jour le `Calendar` avec la nouvelle date (année, mois, jour). On lui passe également la dernière date choisie, en récupérant les valeurs qui se trouvent dans le `Calendar`. `TimePickerDialog`, quant à lui, reçoit un écouteur `OnTimeSetListener` pour mettre à jour la portion horaire du `Calendar` ; on lui passe également la dernière heure choisie et `true` pour indiquer que l'on veut un format sur 24 heures.

Le résultat de cette activité est présenté aux Figures 9.1, 9.2 et 9.3.

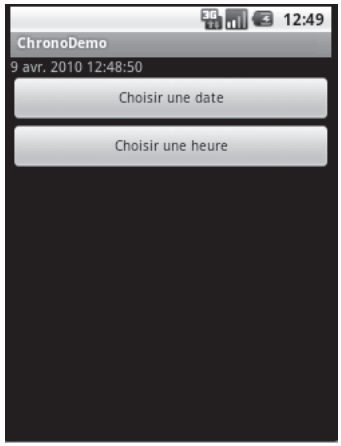


Figure 9.1
L'application `ChronoDemo` *lors de son lancement.*

Figure 9.2
La même application, montrant le dialogue de choix de la date.

Figure 9.3
La même application, montrant le dialogue de choix de l'heure.

Le temps s'écoule comme un fleuve

Pour afficher l'heure sans autoriser les utilisateurs à la modifier, utilisez les widgets `DigitalClock` ou `AnalogClock`. Il suffit simplement de les placer dans votre layout et de les laisser travailler.

Le fichier `main.xml` du projet `Fancy/Clocks` contient ces deux widgets :

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <AnalogClock android:id="@+id/analog"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_alignParentTop="true"
    />
  <DigitalClock android:id="@+id/digital"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_below="@id/analog"
    />
</RelativeLayout>
```

Sans avoir besoin de modifier quoi que ce soit au squelette de code Java produit par `android create project`, on obtient l'application présentée à la Figure 9.4.

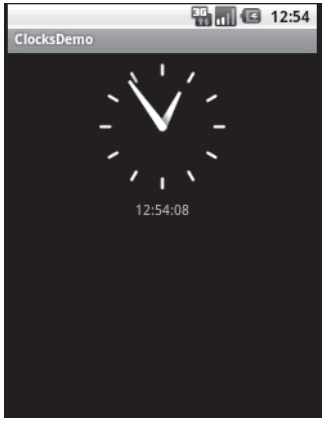


Figure 9.4
L'application `ClocksDemo`.

Si vous recherchez plutôt un chronomètre, le widget `Chronometer` est fait pour vous car il permet de mémoriser le temps écoulé à partir d'un point de départ : il suffit de lui dire quand démarrer (`start()`), s'arrêter (`stop()`) et, éventuellement, de redéfinir le format de la chaîne de texte affichée. La Figure 9.5 montre le résultat obtenu.

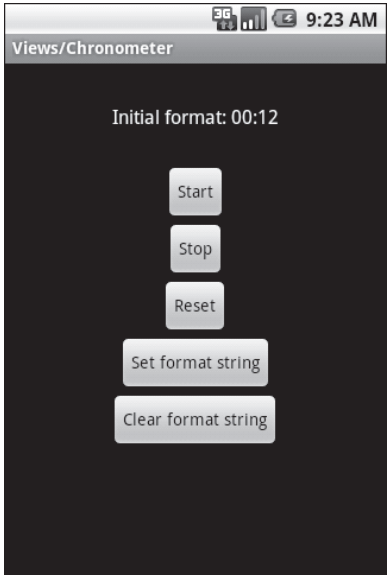


Figure 9.5
La démo de l'API `Views/Chronometer` *du SDK Android 2.0.*

Mesurer la progression

Si une opération doit durer un certain temps, vos utilisateurs doivent pouvoir :

- utiliser un thread en arrière-plan ;
- être tenus au courant de la progression de l'opération, sous peine de penser que l'activité a un problème.

L'approche classique pour tenir les utilisateurs informés d'une progression consiste à utiliser une barre de progression ou un "disque tournant" (pensez à l'animation qui apparaît en haut à droite de la plupart des navigateurs web). Android dispose pour cela du widget `ProgressBar`.

Une `ProgressBar` mémorise la progression, définie par un entier allant de 0 (aucune progression) à une valeur maximale définie par `setMax()` qui indique que l'opération s'est terminée. Par défaut, une barre de progression part de la valeur 0, mais vous pouvez choisir une autre valeur de départ *via* un appel à sa méthode `setProgress()`.

Si vous préférez que la barre soit indéterminée, passez la valeur `true` à sa méthode `setIndeterminate()`.

Dans le code Java, vous pouvez fixer le montant de progression effectuée (*via* `setProgress()`) ou incrémenter la progression courante d'une valeur déterminée (*via* `incrementProgressBy()`). La méthode `getProgress()`, quant à elle, permet de connaître la progression déjà effectuée.

Le widget `ProgressBar` étant intimement lié à l'utilisation des threads – un thread en arrière-plan effectue le traitement et informe de sa progression le thread qui gère l'interface –, nous préférons reporter la démonstration de ce widget jusqu'au Chapitre 15.

Prendre la bonne résolution

La classe `SeekBar` hérite de `ProgressBar` mais, alors que ce dernier est un widget de sortie dont le but consiste à indiquer l'état d'une progression à l'utilisateur, `SeekBar` est un widget d'entrée qui permet de saisir une valeur prise parmi un intervalle donné. La Figure 9.6 montre ce qu'affiche l'exemple fourni avec le SDK d'Android 2.0.

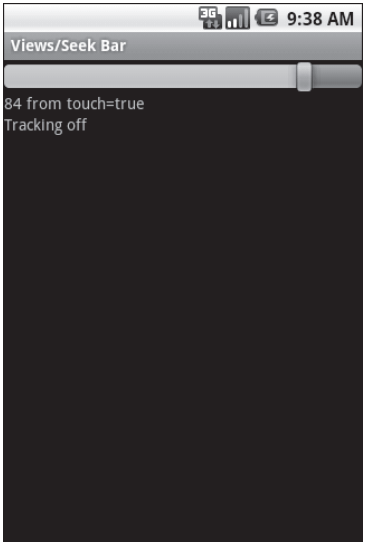


Figure 9.6
La démo de l'API `Views/SeekBar` *du SDK Android 2.0.*

L'utilisateur peut faire glisser le curseur ou cliquer sur l'un des deux côtés pour le repositionner. Ce curseur désigne en réalité une valeur dans un intervalle déterminé – 0 à 100 par défaut, mais vous pouvez modifier la borne supérieure par un appel à `setMax()`. La méthode `getProgress()` renvoie la position courante et l'enregistrement d'un écouteur à

l'aide de `setOnSeekBarChangeListener()` permet d'être prévenu de toute modification du curseur.

L'exemple `RatingBar` que nous avons étudié au Chapitre 8 était une variante de ce widget.

Utilisation d'onglets

La philosophie générale d'Android consiste à faire en sorte que les activités soient courtes et agréables. Lorsqu'il y a plus d'informations que ne peut raisonnablement en contenir l'écran (bien que l'on puisse utiliser des barres de défilement), il peut être préférable de produire ces informations supplémentaires par une autre activité, lancée *via* une *Intent*, comme on l'explique au Chapitre 18. Cependant, ceci peut être assez compliqué à mettre en place. En outre, il arrive parfois que l'on doive recueillir et traiter beaucoup d'informations en une seule opération.

Dans une interface graphique classique, nous pourrions utiliser des onglets à cette fin, comme un `JTabbedPane` en Java/Swing. Avec Android, on dispose désormais d'un conteneur `TabHost` qui fonctionne exactement de la même façon – une portion de ce qu'affiche l'activité est liée à des onglets qui, lorsque l'on clique dessus, permettent de passer d'une partie de la vue à une autre. Une activité utilisera par exemple un onglet pour la saisie d'un emplacement et un autre pour afficher cet emplacement sur une carte.

Certains kits de développement considèrent simplement les onglets comme des objets cliquables, permettant de passer d'une vue à l'autre. D'autres les considèrent comme une combinaison de l'élément cliquable et du contenu qui apparaît lorsque l'on clique dessus. Avec Android, les boutons et les contenus des onglets étant des entités distinctes, nous emploierons les termes "bouton de l'onglet" et "contenu de l'onglet" dans cette section.

Les pièces du puzzle

Pour mettre en place des onglets dans une vue, vous avez besoin des widgets et des conteneurs suivants :

- `TabHost` est le conteneur général pour les boutons et les contenus des onglets.
- `TabWidget` implémente la ligne des boutons des onglets, qui contient les labels et, éventuellement, des icônes.
- `FrameLayout` est le conteneur des contenus des onglets : chaque contenu d'onglet est un fils du `FrameLayout`.

Cette approche ressemble à celle de XUL, utilisée par Mozilla : les éléments `tabbox`, `tabs` et `tabpanels` de XUL correspondent respectivement à `TabHost`, `TabWidget` et `FrameLayout`.

Idiosyncrasies

La version actuelle d'Android exige de respecter les règles suivantes pour que ces trois composants puissent fonctionner de concert :

- L'identifiant `android:id` du `TabWidget` doit être `@android:id/tabs`.
- Si vous souhaitez utiliser `TabActivity`, l'identifiant `android:id` du `TabHost` doit être `@android:id/tabhost`.
- `TabActivity`, comme `ListActivity`, enveloppe un motif d'interface graphique classique (une activité composée entièrement d'onglets) dans une sous-classe d'activité. Vous n'en avez pas nécessairement besoin – une activité classique peut très bien utiliser également des onglets.

Voici le fichier de description d'une activité utilisant des onglets, tiré du projet `Fancy/Tab` :

```
<?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">
  <TabHost android:id="@+id/tabhost"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TabWidget android:id="@android:id/tabs"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
    />
    <FrameLayout android:id="@android:id/tabcontent"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:paddingTop="62px">
      <AnalogClock android:id="@+id/tab1"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:layout_centerHorizontal="true"
      />
      <Button android:id="@+id/tab2"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:text="Bouton semi-aleatoire"
      />
    </FrameLayout>
  </TabHost>
</LinearLayout>
```

Vous remarquerez que les éléments `TabWidget` et `FrameLayout` sont des fils directs de `TabHost` et que l'élément `FrameLayout` a lui-même un fils représentant les différents onglets. Ici, il y en a deux : une horloge et un bouton. Dans un scénario plus compliqué, les onglets seraient regroupés dans un conteneur (un `LinearLayout`, par exemple) avec leurs propres contenus.

Code Java

Le code Java doit indiquer au `TabHost` quelles sont les vues qui représentent les contenus des onglets et à quoi doivent ressembler les boutons de ces onglets. Tout ceci est encapsulé dans des objets `TabSpec`. On récupère une instance de `TabSpec` *via* la méthode `newTabSpec()` du `TabHost`, on la remplit puis on l'ajoute au `TabHost` dans le bon ordre.

Les deux méthodes essentielles de `TabSpec` sont les suivantes :

- `setContent()`, qui permet d'indiquer le contenu de cet onglet. Généralement, il s'agit de l'identifiant `android:id` de la vue que l'on veut montrer lorsque l'onglet est choisi.
- `setIndicator()`, qui permet de fournir le titre du bouton de l'onglet. Cette méthode est surchargée pour permettre de fournir également un objet `Drawable` représentant l'icône de l'onglet.

Notez que les "indicateurs" des onglets peuvent, en fait, être eux-mêmes des vues, ce qui permet de faire mieux qu'un simple label et une icône facultative.

Notez également que vous devez appeler la méthode `setup()` de l'objet `TabHost` avant de configurer les objets `TabSpec`. Cet appel n'est pas nécessaire si votre activité dérive de la classe de base `TabActivity`.

Voici, par exemple, le code Java permettant de faire fonctionner les onglets de la section précédente :

```
package com.commonsware.android.fancy;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TabHost;

public class TabDemo extends Activity {
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    TabHost tabs=(TabHost)findViewById(R.id.tabhost);
```

```
        tabs.setup();

        TabHost.TabSpec spec=tabs.newTabSpec("tag1");

        spec.setContent(R.id.tab1);
        spec.setIndicator("Heure");
        tabs.addTab(spec);

        spec=tabs.newTabSpec("tag2");
        spec.setContent(R.id.tab2);
        spec.setIndicator("Bouton");
        tabs.addTab(spec);

        tabs.setCurrentTab(0);
    }
}
```

On retrouve notre `TabHost` *via* un appel à la méthode `findViewById()`, puis l'on appelle sa méthode `setup()`.

Ensuite, on crée une instance de `TabSpec` *via* un appel à `newTabSpec()` auquel on passe un marqueur dont le but est encore inconnu. On appelle les méthodes `setContent()` et `setIndicator()` de cette instance, puis la méthode `addTab()` de l'objet `TabHost` pour lui ajouter le `TabSpec`. Enfin, on choisit l'onglet qui s'affichera initialement, à l'aide de la méthode `setCurrentTab()` (la valeur 0 désigne le premier onglet).

Les Figures 9.7 et 9.8 montrent ce qu'affiche cette application.

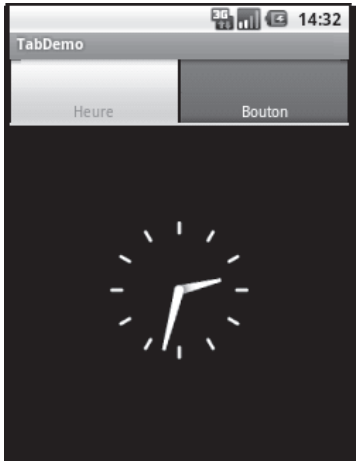


Figure 9.7
L'application `TabDemo` *affichant son premier onglet.*

Figure 9.8
La même application affichant son second onglet.

Ajouts dynamiques

`TabWidget` est configuré pour simplifier la définition des onglets au moment de la compilation. Cependant, vous voudrez parfois créer des onglets au cours de l'exécution de votre activité. Imaginons, par exemple, un client de courrier où les différents e-mails s'ouvrent dans leurs propres onglets, afin de faciliter le passage d'un message à l'autre. Dans cette situation, vous ne pouvez pas savoir à l'avance le nombre d'onglets ni leur contenu : vous devez attendre que l'utilisateur ouvre un message de courrier.

Heureusement, Android permet également d'ajouter dynamiquement des onglets en cours d'exécution.

Cet ajout fonctionne exactement comme on vient de le voir, sauf qu'il faut utiliser une autre variante de `setContent()` qui prend en paramètre une instance de `TabHost.TabContentFactory` qui est simplement une méthode de rappel qui sera appelée automatiquement : il suffit de fournir une implémentation de `createTabContent()` et de l'utiliser pour construire et renvoyer la vue qui deviendra le contenu de l'onglet.

Cette approche est présentée dans le projet `Fancy/DynamicTab`.

La description de l'interface de l'activité met en place les onglets et n'en définit qu'un seul, contenant un simple bouton :

```
<?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">
    <TabHost android:id="@+id/tabhost"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent">
      <TabWidget android:id="@android:id/tabs"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
      />
      <FrameLayout android:id="@android:id/tabcontent"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:paddingTop="62px">
        <Button android:id="@+id/buttontab"
          android:layout_width="fill_parent"
          android:layout_height="fill_parent"
          android:text="Bouton semi-aleatoire"
        />
      </FrameLayout>
    </TabHost>
  </LinearLayout>
```

Nous voulons maintenant ajouter de nouveaux onglets à mesure qu'on clique sur ce bouton, ce qui se réalise en quelques lignes de code :

```
public class DynamicTabDemo extends Activity {
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    final TabHost tabs=(TabHost)findViewById(R.id.tabhost);

    tabs.setup();

    TabHost.TabSpec spec=tabs.newTabSpec("buttontab");
    spec.setContent(R.id.buttontab);
    spec.setIndicator("Bouton");
    tabs.addTab(spec);

    tabs.setCurrentTab(0);
```

```
        Button btn=(Button)tabs.getCurrentView().findViewById(R.id.buttontab);

        btn.setOnClickListener(new View.OnClickListener() {
          public void onClick(View view) {
            TabHost.TabSpec spec=tabs.newTabSpec("tag1");

            spec.setContent(new TabHost.TabContentFactory() {
              public View createTabContent(String tag) {
                return(new AnalogClock(DynamicTabDemo.this));
              }
            });
            spec.setIndicator("Heure");
            tabs.addTab(spec);
          }
        });
      }
    }
```

On crée un objet `TabHost.TabSpec` dans la méthode de rappel `setOnClickListener()` de notre bouton en lui passant en paramètre une fabrique `TabHost.TabContentFactory` anonyme. Cette fabrique, à son tour, renvoie la vue qui sera utilisée pour l'onglet – ici un `AnalogClock`. Le code de construction de cette vue pourrait être bien plus élaboré et utiliser, par exemple, un `LayoutInflater` pour créer une vue à partir d'un fichier de description XML.

La Figure 9.9 montre que l'activité n'affiche qu'un seul onglet lorsqu'elle est lancée. La Figure 9.10 montre plusieurs onglets, créés en cours d'exécution.

Figure 9.9
L'application `DynamicTabDemo` *avec son unique onglet initial.*

Figure 9.10
La même application, après la création de trois onglets en cours d'exécution.

Intent et *View*

Dans les exemples précédents, le contenu de chaque onglet était une `View` : un `Button`, par exemple. Ce type de configuration est simple à mettre en place, mais ce n'est pas le seul : vous pouvez également intégrer une autre activité à partir de votre application *via* une `Intent`.

Les `Intent` permettent de préciser ce que vous voulez réaliser, puis de demander à Android de trouver un moyen de l'accomplir, ce qui force souvent l'activité à se multiplier. À chaque fois que vous lancez une application à partir du lanceur d'applications d'Android, par exemple, ce lanceur crée une `Intent` et laisse Android ouvrir l'activité associée. Ce concept, ainsi que le placement des activités dans des onglets, sera décrit au Chapitre 18.

Tout faire basculer

Parfois, on souhaite bénéficier de l'avantage des onglets (ne voir que certaines vues à la fois) sans pour autant utiliser leur présentation graphique (parce que, par exemple, les onglets prennent trop de place à l'écran). On peut ainsi préférer passer d'une vue à l'autre par un mouvement du doigt sur l'écran ou en secouant le terminal.

La bonne nouvelle est que le mécanisme interne des onglets pour basculer entre les vues est disponible dans le conteneur `ViewFlipper`, qui peut être utilisé différemment d'un onglet traditionnel.

`ViewFlipper` hérite de `FrameLayout`, que nous avons utilisé plus haut pour décrire le fonctionnement interne d'un `TabWidget`. Cependant, il ne montre que la première vue fille au départ : c'est à vous qu'il appartient de mettre en place le basculement entre les vues, soit manuellement par une action de l'utilisateur, soit automatiquement par un timer.

Voici, par exemple, le fichier de description du projet `Fancy/Flipper1`, qui utilise un `Button` et un `ViewFlipper` :

```
<?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/flip_me"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="Bascule-moi !"
      />
  <ViewFlipper android:id="@+id/details"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <TextView
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:textStyle="bold"
      android:textColor="#FF00FF00"
      android:text="Premier panneau"
    />
    <TextView
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:textStyle="bold"
      android:textColor="#FFFF0000"
      android:text="Second panneau"
    />
    <TextView
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
```

```
        android:textStyle="bold"
        android:textColor="#FFFFFF00"
        android:text="Troisieme panneau"
      />
    </ViewFlipper>
  </LinearLayout>
```

Ce layout définit trois vues filles de `ViewFlipper`, chacune étant un `TextView` contenant un simple message. Vous pourriez évidemment choisir des vues plus complexes.

Basculement manuel

Pour basculer manuellement entre les vues, nous devons ajouter un écouteur au bouton pour que le basculement ait lieu lorsqu'on clique dessus :

```
public class FlipperDemo extends Activity {
  ViewFlipper flipper;

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

    flipper=(ViewFlipper)findViewById(R.id.details);

    Button btn=(Button)findViewById(R.id.flip_me);

    btn.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        flipper.showNext();
      }
    });
  }
}
```

Il s'agit simplement d'appeler la méthode `showNext()` de `ViewFlipper`, comme pour n'importe quelle classe `ViewAnimator`.

Le résultat est une activité très simple : un clic sur le bouton fait apparaître le `TextView` suivant, en rebouclant sur le premier lorsqu'ils se sont tous affichés (voir Figures 9.11 et 9.12).

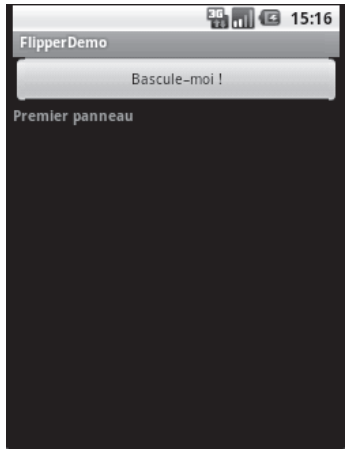


Figure 9.11
L'application `Flipper1` *montant le premier panneau.*

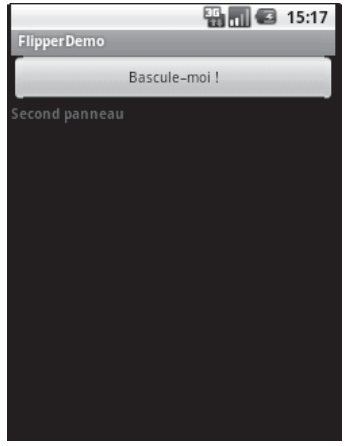


Figure 9.12
La même application, après basculement vers le second panneau.

Nous pourrions bien sûr gérer tout cela plus simplement en utilisant un seul `TextView` et en modifiant son texte et sa couleur à chaque clic. Cependant, vous pouvez imaginer que le contenu du `ViewFlipper` pourrait être bien plus compliqué – inclure, par exemple, tout ce que l'on peut mettre dans un `TabView`.

Ajout de contenu à la volée

Comme pour un `TabWidget`, le contenu d'un `ViewFlipper` peut ne pas être connu lors de la compilation et, comme pour un `TabWidget`, il est relativement simple d'ajouter du contenu à la volée.

Voici, par exemple, le layout d'une autre activité, celle du projet `Fancy/Flipper2` :

```
<?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"
    >
  <ViewFlipper android:id="@+id/details"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
  </ViewFlipper>
</LinearLayout>
```

Vous remarquerez que l'élément `ViewFlipper` n'a aucun contenu au moment de la compilation. Notez également qu'il n'y a pas de bouton pour basculer entre les contenus – nous reviendrons sur ce point dans un instant.

Pour le contenu du `ViewFlipper`, nous créerons de gros boutons contenant, chacun, un ensemble de mots quelconques. Nous configurerons également le `ViewFlipper` pour qu'il boucle automatiquement sur ces widgets, en utilisant une animation pour la transition :

```
public class FlipperDemo2 extends Activity {
  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"};
  ViewFlipper flipper;

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

    flipper=(ViewFlipper)findViewById(R.id.details);

    flipper.setInAnimation(AnimationUtils.loadAnimation(this,
                                        R.anim.push_left_in));
```

```
        flipper.setOutAnimation(AnimationUtils.loadAnimation(this,
                                                R.anim.push_left_out));
        for (String item : items) {
          Button btn=new Button(this);

          btn.setText(item);

          flipper.addView(btn,
                          new ViewGroup.LayoutParams(
                                  ViewGroup.LayoutParams.FILL_PARENT,
                                  ViewGroup.LayoutParams.FILL_PARENT));
        }

        flipper.setFlipInterval(2000);
        flipper.startFlipping();
      }
    }
```

Après avoir obtenu le widget `ViewFlipper` à partir du fichier de disposition, nous commençons par configurer les animations d'entrée et de sortie. En termes Android, une animation est une description de la sortie d'un widget ("out") et de son entrée ("in") dans la zone visible de l'écran. Les animations sont des ressources stockées dans le répertoire `res/anim/` du projet. Dans cet exemple, nous utilisons deux exemples fournis par le SDK et publiés sous les termes de la licence Apache 2.0. Comme leur nom l'indique, les widgets sont "poussés" à gauche pour entrer ou sortir de la zone visible.

Les animations sont un sujet complexe que je traite dans le livre The Busy Coder's Guide to Advanced Android Development *(CommonsWare LLC, 2009).*

Basculement automatique

Après avoir parcouru tous les mots en les transformant en autant de boutons fils de l'objet `ViewFlipper`, nous configurons ce dernier pour qu'il bascule automatiquement entre ses fils (`flipper.setFlipInterval(2000);`) et nous lançons le basculement (`flipper.startFlipping();`).

Le résultat est une suite sans fin de boutons apparaissant puis disparaissant vers la gauche au bout de 2 secondes, en étant remplacés à chaque fois par le bouton suivant de la séquence. L'ensemble revient au premier bouton après la disparition du dernier (voir Figure 9.13).

Figure 9.13
L'application `Flipper2`*,*
avec une transition animée.

Ce basculement automatique est utile pour les panneaux d'information ou les autres situations dans lesquelles vous voulez afficher beaucoup d'informations dans un espace réduit. Ces différentes vues basculant automatiquement de l'une à l'autre, il serait risqué de demander aux utilisateurs d'interagir avec elles – une vue pourrait disparaître au milieu d'une interaction.

Fouiller dans les tiroirs

Depuis longtemps, les développeurs Android réclamaient un conteneur de type tiroir, fonctionnant comme celui de l'écran d'accueil, qui contient les icônes pour lancer les applications. L'implémentation officielle existait dans le code open-source mais n'était pas intégrée dans le SDK... jusqu'à Android 1.5, qui a fourni le widget `SlidingDrawer`.

À la différence de la plupart des autres conteneurs, `SlidingDrawer` change d'aspect puisqu'il passe d'une position fermée à une position ouverte. Cette caractéristique implique quelques restrictions sur le conteneur dans lequel peut se trouver le `SlidingDrawer` puisqu'il doit permettre à plusieurs widgets de cohabiter les uns au-dessus des autres. `RelativeLayout` et `FrameLayout` satisfont cette exigence – `FrameLayout` est un conteneur conçu spécialement pour empiler les widgets les uns sur les autres. `LinearLayout`, en revanche, ne permet pas d'empiler des widgets (ils sont placés les uns après les autres, en ligne ou en colonne) –, c'est la raison pour laquelle un `SlidingDrawer` ne doit pas être un fils direct d'un élément `LinearLayout`.

Voici un exemple tiré du projet `Fancy/DrawerDemo`, avec un `SlidingDrawer` placé dans un `FrameLayout` :

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#FF4444CC"
    >
  <SlidingDrawer
    android:id="@+id/drawer"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:handle="@+id/handle"
    android:content="@+id/content">
    <ImageView
      android:id="@id/handle"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:src="@drawable/tray_handle_normal"
    />
    <Button
      android:id="@id/content"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:text="Je suis dedans !"
    />
  </SlidingDrawer>
</FrameLayout>
```

`Le SlidingDrawer doit contenir :`

- une poignée – le plus souvent un `ImageView`, comme ici ;
- le contenu du tiroir lui-même – généralement un conteneur, bien qu'ici nous utilisions un bouton.

En outre, `SlidingDrawer` doit connaître les valeurs `android:id` de la poignée et du contenu en les stockant, respectivement, dans ses attributs `android:handle` et `android:content`. Ceci permet au tiroir de savoir comment s'animer lorsqu'il s'ouvre ou se ferme.

La Figure 9.14 montre l'aspect du tiroir fermé, avec la poignée qu'on lui a fournie. La Figure 9.15 montre le tiroir ouvert, avec son contenu.

Figure 9.14
L'application `DrawerDemo` *avec son tiroir fermé.*

Figure 9.15
La même application avec le tiroir ouvert.

Comme on pourrait s'y attendre, on peut ouvrir et refermer le tiroir à partir du code Java ou par des événements de "touchés" utilisateurs (ceux-ci sont gérés par le widget et vous n'avez donc pas à vous en occuper). Il existe deux groupes de méthodes : les premières agissent instantanément (`open()`, `close()` et `toggle()`), les autres utilisent une animation (`animateOpen()`, `animateClose()` et `animateToggle()`).

Le tiroir se verrouille avec `lock()` et se déverrouille avec `unlock()` ; lorsqu'il est verrouillé, le tiroir ne répond pas aux touchés sur l'écran.

Vous pouvez également, si vous le souhaitez, enregistrer trois types de méthodes de rappel :

- un écouteur qui sera appelé lors de l'ouverture du tiroir ;

- un écouteur qui sera appelé lors de la fermeture du tiroir ;
- un écouteur qui sera appelé lorsque le tiroir "défile" (c'est-à-dire lorsque l'utilisateur tire ou repousse la poignée).

Le `SlidingDrawer` du lanceur, par exemple, change l'icône de sa poignée pour qu'elle signifie "ouvrir", "fermer" ou "supprimer" (lorsque l'on touche pendant un certain temps une icône du bureau). Pour ce faire, il utilise notamment des méthodes de rappel comme celles que nous venons de citer.

`SlidingDrawer` peut être vertical ou horizontal. Cependant, cette orientation reste identique quelle que soit celle de l'écran : en d'autres termes, si vous faites pivoter le terminal ou l'émulateur pendant qu'il exécute `DrawerDemo`, le tiroir s'ouvrira toujours en partant du bas – il ne "colle" pas toujours au bord par rapport à celui auquel il s'est ouvert. Pour que le tiroir s'ouvre toujours du même côté, comme le lanceur, vous aurez besoin de layouts différents pour le mode portrait et le mode paysage – un sujet que nous aborderons au Chapitre 20.

Autres conteneurs intéressants

Android fournit également le conteneur `AbsoluteLayout`, dont le contenu est disposé en fonction de coordonnées spécifiques – on lui indique où placer un fils en précisant ses coordonnées X, Y, et Android le positionne à cet endroit sans poser de question.

Ceci a l'avantage de fournir un positionnement précis ; en revanche, cela signifie également que les vues n'auront un aspect correct que sur des écrans d'une certaine dimension, à moins d'écrire beaucoup de code pour ajuster les coordonnées en fonction de la taille de l'écran. Les écrans Android pouvant avoir n'importe quelle taille et ces tailles évoluant continuellement, l'utilisation d'`AbsoluteLayout` risque de devenir assez problématique.

Officiellement, `AbsoluteLayout` *est considéré comme obsolète : bien qu'il reste disponible, il est désormais déconseillé de l'utiliser.*

Android dispose également d'une nouvelle variante de liste, `ExpandableListView`, qui fournit une représentation arborescente simplifiée autorisant deux niveaux de profondeur : les groupes et les fils. Les groupes contiennent les fils, qui sont les "feuilles" de l'arbre. Cette liste nécessite un nouvel ensemble d'adaptateurs car la famille `ListAdapter` ne fournit aucune information de groupage pour les éléments d'une liste.

10

Le framework des méthodes de saisie

Android 1.5 a introduit le framework des méthodes de saisie IMF (*Input Method Framework*), que l'on désigne sous le terme de *claviers logiciels* bien qu'il ne soit pas nécessairement tout à fait exact car IMF pourrait être utilisé pour la reconnaissance d'écriture ou tout autre moyen de saisir du texte *via* l'écran.

Ce chapitre explique comment utiliser IMF pour adapter les claviers logiciels aux besoins de votre application.

Claviers physiques et logiciels

Certains terminaux Android comme le HTC Magic n'ont pas de clavier physique ; d'autres, comme le G1, ont un clavier physique qui n'est pas disponible en permanence (il faut l'ouvrir). Il est probable que, dans le futur, apparaîtront des terminaux Android avec un clavier

physique permanent (comme les netbooks et certains téléphones mobiles actuels, qui ont toujours un clavier sous leur écran). IMF sait gérer tous ces scénarios : s'il n'y a pas de clavier physique, un IME (*Input Method Editor*) apparaîtra lorsque l'utilisateur tape sur un `EditText`.

Vos applications n'ont pas besoin d'être modifiées du moment que la fonctionnalité par défaut de l'IME vous convient. Heureusement, Android est suffisamment malin pour deviner ce que vous voulez : vous pouvez donc faire des tests avec l'IME, mais sinon il n'est pas nécessaire d'effectuer des modifications spécifiques dans votre code.

Le clavier peut aussi ne pas se comporter tout à fait comme vous le souhaitez. Dans le projet `Basic/Field`, par exemple, l'IME recouvre la zone de saisie de l'activité `FieldDemo`, comme on le voit à la Figure 10.1 : il serait préférable de pouvoir mieux contrôler la façon dont l'IME apparaît et réagit. Heureusement, nous allons voir que le framework vous laisse une grande latitude.

Figure 10.1
L'IME de l'application `FieldDemo`.

Adaptation à vos besoins

Android 1.1 et les versions précédentes offraient de nombreux attributs pour les widgets `EditText` afin de contrôler leur style de saisie – `android:password`, par exemple, pour indiquer qu'un champ sert à saisir un mot de passe (et que le texte entré doit donc être caché des yeux indiscrets). Avec Android 1.5 et IMF, la plupart ont été regroupés dans un seul attribut `android:inputType` dont la valeur est composée d'une classe et de modificateurs, séparés par le caractère "pipe" (`|`). La classe décrit généralement ce que l'utilisateur est autorisé à

saisir et détermine l'ensemble de base des touches disponibles sur le clavier logiciel. Les classes possibles sont les suivantes :

- `text` (par défaut) ;
- `number` ;
- `phone` ;
- `datetime` ;
- `date` ;
- `time`.

La plupart d'entre elles offrent un ou plusieurs modificateurs pour affiner ce que pourra saisir l'utilisateur. Pour mieux comprendre leur signification, étudions le fichier `res/layout/main.xml` du projet `InputMethod/IMEDemo1` :

```
<?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="Aucune regle speciale : "
    />
    <EditText />
  </TableRow>
  <TableRow>
    <TextView
      android:text="Adresse e-mail : "
    />
    <EditText
      android:inputType="text|textEmailAddress"
    />
  </TableRow>
  <TableRow>
    <TextView
      android:text="Nombre decimal signe : "
    />
    <EditText
      android:inputType="number|numberSigned|numberDecimal"
    />
```

```
    </TableRow>
    <TableRow>
      <TextView
        android:text="Date :"
      />
      <EditText
        android:inputType="date"
      />
    </TableRow>
    <TableRow>
      <TextView
        android:text="Texte multilignes : "
      />
      <EditText
        android:inputType="text|textMultiLine|textAutoCorrect"
        android:minLines="3" android:gravity="top"
      />
    </TableRow>
  </TableLayout>
```

Nous utilisons ici un conteneur `TableLayout` de cinq lignes utilisant, chacune, une variante légèrement différente d'`EditText` :

- La première n'utilise aucun attribut.
- La deuxième utilise `android:inputType = "text|textEmailAddress"`, ce qui signifie que c'est une saisie de texte, une adresse de courrier électronique, plus précisément.
- La troisième permet de saisir un nombre décimal signé grâce à `android:inputType = "number|numberSigned|numberDecimal"`.
- La quatrième est configurée pour la saisie d'une date (`android:inputType = "date"`).
- La dernière autorise une saisie sur plusieurs lignes, avec autocorrection des erreurs d'orthographe (`android:inputType = "text|textMultiLine|textAutoCorrect"`).

La classe et les modificateurs permettent d'adapter le clavier à vos besoins : un champ de saisie de texte brut produira donc un clavier logiciel de base, comme le montre la Figure 10.2.

Figure 10.2
Clavier logiciel standard.

Un champ de saisie d'adresse e-mail ajoute les touches `@` et `.com` au clavier, au prix d'une barre d'espace plus petite, comme le montre la Figure 10.3.

Figure 10.3
Clavier logiciel pour la saisie des adresses e-mail.

La Figure 10.4 montre que les champs numériques et date limitent les touches au clavier numérique et à un ensemble de symboles autorisés pour un champ donné.

Figure 10.4
Clavier logiciel pour les nombres décimaux signés.

Ce ne sont que quelques exemples qui vous montrent qu'en choisissant la valeur `android:inputType` appropriée vous pouvez offrir aux utilisateurs le clavier qui convient le mieux à ce qu'ils doivent saisir.

Dire à Android où aller

Vous aurez peut-être noté une subtile différence entre les claviers de la Figure 10.2 et celui de la Figure 10.4 – hormis l'ajout des touches `@` et `.com` : la touche `Suivant` est venue remplacer à la Figure 10.4 la touche `Entrée` de la Figure 10.2. Ceci signifie deux choses :

- Les widgets `EditText` sont multilignes par défaut si vous n'utilisez pas `android:inputType`.
- Vous pouvez contrôler ce qui s'affiche sur cette touche inférieure droite, appelée *bouton accessoire*.

Par défaut, le bouton accessoire d'un `EditText` avec un attribut `android:inputType` sera `Suivant` pour vous permettre de passer au champ `EditText` suivant ou `OK` si vous vous trouvez sur le dernier `EditText` de l'écran. Vous pouvez toutefois indiquer manuellement ce qui s'affichera sur ce bouton à l'aide de l'attribut `android:imeOptions`. Dans le fichier `res/layout/main.xml` du projet `InputMethod/IMEDemo2`, par exemple, deux champs de saisie précisent l'apparence du bouton accessoire :

```
<?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="fill_parent">

        <TableLayout
          android:layout_width="fill_parent"
          android:layout_height="fill_parent"
          android:stretchColumns="1" >

          <TableRow>
            <TextView
              android:text="Aucune regle speciale : "
            />
            <EditText />
          </TableRow>
          <TableRow>
            <TextView
              android:text="Adresse e-mail : "
            />
            <EditText
              android:inputType="text|textEmailAddress"
              android:imeOptions="actionSend"
            />
          </TableRow>
          <TableRow>
            <TextView
              android:text="Nombre decimal signe : "
            />
            <EditText
              android:inputType="number|numberSigned|numberDecimal"
              android:imeOptions="actionDone"
            />
          </TableRow>
          <TableRow>
            <TextView
              android:text="Date:"
            />
            <EditText
              android:inputType="date"
            />
          </TableRow>
          <TableRow>
```

```
        <TextView
          android:text="Texte multilignes :"
        />
        <EditText
          android:inputType="text|textMultiLine|textAutoCorrect"
          android:minLines="3"
          android:gravity="top"
        />
      </TableRow>
    </TableLayout>
  </ScrollView>
```

Ici, nous attachons une action `Envoyer` au bouton accessoire de la saisie d'une adresse e-mail (`android:imeOptions = "actionSend"`) et l'action `OK` à celui de la saisie d'un nombre (`android:imeOptions = "actionDone"`).

Par défaut, `Suivant` déplace le focus sur le widget `EditText` suivant et `OK` ferme l'IME. Cependant, pour ces actions ou pour toutes celles comme `Envoyer`, vous pouvez utiliser `setOnEditorActionListener()` sur l'`EditText` afin de savoir quand le bouton accessoire ou la touche Entrée sont cliqués. Vous recevrez alors un indicateur signalant l'action voulue (`IME_ACTION_SEND`, par exemple) et vous pourrez traiter cette requête (envoyer un courrier à l'adresse e-mail qui a été saisie, par exemple).

Mise en place

Vous noterez les différences entre le layout de `IMEDemo1` et celui de `IMEDemo2` : dans ce dernier, nous utilisons un conteneur `ScrollView` pour envelopper le `TableLayout`. Ceci nous amène à un autre niveau de contrôle sur les IME : que devient le layout de notre activité lorsque le clavier apparaît ? En fonction des circonstances, il y a trois possibilités :

- Android peut faire "glisser" vers le haut toute l'activité afin de laisser la place au clavier ; il peut également recouvrir celle-ci selon que l'`EditText` en cours de saisie se trouve en haut ou en bas. Ceci a donc pour effet de masquer certaines parties de votre interface.
- Il peut modifier la taille de l'activité afin qu'elle tienne dans une zone d'écran plus réduite pour permettre à l'IME de s'afficher en dessous. Cette technique est parfaite lorsque le layout peut être réduit (lorsqu'il est dominé par une liste ou un champ de saisie multilignes qui n'a pas besoin de tout l'écran pour être fonctionnel, par exemple).

- En mode paysage, Android peut afficher l'IME sur tout l'écran et masquer ainsi l'intégralité de l'activité. Ceci permet de disposer d'un clavier plus grand pour faciliter la saisie.

Android contrôle tout seul l'option plein écran et, par défaut, choisira le mode défilement ou réduction en fonction de votre layout. L'attribut `android:windowSoftInputMode` de l'élément `<activity>` du fichier `AndroidManifest.xml` vous permet toutefois de fixer vous-même ce mode. Voici, par exemple, le manifeste de `IMEDemo2` :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.commonsware.android.imf.two"
         android:versionCode="1"
         android:versionName="1.0">
  <application android:label="@string/app_name"
               android:icon="@drawable/cw">
    <activity android:name=".IMEDemo2"
              android:label="@string/app_name"
              android:windowSoftInputMode="adjustResize">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
       <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Figure 10.5
Layout réduit, avec une barre de défilement.

Libérez le Dvorak qui est en vous

Vous pouvez créer et distribuer vos propres IME : un clavier Dvorak, un clavier pour une autre langue ou un clavier qui répète oralement les touches pressées, par exemple.

Un IME est assemblé sous la forme d'un service, c'est-à-dire un composant Android que nous décrirons aux Chapitres 29 et 30. Si vous voulez créer un clavier, étudiez l'application `SoftKeyboard` distribuée avec le SDK Android 1.5 et, évidemment, le code source d'Android lui-même (recherchez la classe `LatinIME`).

11

Utilisation des menus

Les activités Android peuvent comprendre des menus, telles les applications de bureau et de certains systèmes mobiles comme Windows Mobile. Android les appelle *menus d'options*. Certains terminaux Android disposent même d'une touche dédiée à leur ouverture ; d'autres offrent des moyens différents pour les faire apparaître : la tablette Android Archos 5, par exemple, se sert d'un bouton sur l'écran.

En outre, vous pouvez créer des *menus contextuels,* comme dans la plupart des kits de développement graphiques. Dans une interface utilisateur classique, ce type de menu s'ouvre en cliquant sur le bouton droit de la souris ; sur les terminaux mobiles, ils apparaissent généralement lorsque l'utilisateur touche un widget pendant un certain temps. Si un `TextView` dispose d'un menu contextuel et que le terminal a un écran tactile, par exemple, vous pouvez toucher le label pendant une seconde ou deux : un menu surgira alors pour vous permettre de choisir l'une des entrées du menu.

Menus d'options

Un menu d'option se déclenche en appuyant sur le bouton Menu du terminal et a deux modes de fonctionnement : icône et étendu. Lorsque l'utilisateur appuie sur le bouton

Menu, le menu est en mode icône et n'affiche que les six premiers choix sous la forme de gros boutons faciles à sélectionner, disposés en ligne en bas de l'écran. Si ce menu compte plus de six choix, le sixième bouton affiche "Plus" – cliquer sur cette option fait passer le menu en mode étendu, qui affiche tous les choix restants. L'utilisateur peut bien sûr faire défiler le menu afin d'effectuer n'importe quel choix.

Création d'un menu d'options

Au lieu de construire le menu d'options de votre activité dans `onCreate()`, comme le reste de votre interface graphique, vous devez implémenter la méthode de rappel `onCreateOptionsMenu()` en lui passant une instance de la classe `Menu`.

La première opération à réaliser consiste à établir un chaînage vers la méthode de la superclasse (*via* `super.onCreateOptionsMenu(menu)`), afin qu'Android puisse lui ajouter les choix nécessaires. Puis vous pouvez ajouter les vôtres, comme on l'expliquera bientôt.

Si vous devez modifier le menu au cours de l'activité (pour, par exemple, désactiver un choix devenu inadéquat), il suffit de manipuler l'instance de `Menu` que vous avez transmise à `onCreateOptionsMenu()` ou d'implémenter la méthode `onPrepareOptionsMenu()`, qui est appelée avant chaque affichage du menu.

Ajout de choix et de sous-menus

Pour ajouter des choix au menu, utilisez la méthode `add()`. Celle-ci est surchargée pour pouvoir recevoir des combinaisons des paramètres suivants :

- Un identifiant de groupe (un entier) qui doit valoir `NONE`, sauf si vous créez un ensemble d'options de menu utilisable avec `setGroupCheckable()` (voir plus loin).
- Un identifiant de choix (également un entier) servant à identifier la sélection dans la méthode de rappel `onOptionsItemSelected()` lorsqu'une option du menu a été choisie.
- Un identifiant d'ordre (encore un entier), indiquant l'emplacement du choix dans le menu lorsque ce dernier contient des options ajoutées par Android – pour l'instant, contentez-vous d'utiliser `NONE`.
- Le texte du choix, sous la forme d'une `String` ou d'un identifiant de ressource.

Toutes les méthodes `add()` renvoient une instance de `MenuItem` qui vous permet ensuite de modifier tous les réglages du choix concerné (son texte, par exemple).

Vous pouvez également mettre en place un raccourci pour un choix – un mnémonique d'un seul caractère, permettant de sélectionner ce choix lorsque le menu est visible. Android permet d'utiliser des raccourcis alphabétiques et numériques, mis en place respectivement

par `setAlphabeticShortcut()` et `setNumericShortcut()`. Le menu est placé en mode raccourci alphabétique en passant le paramètre `true` à sa méthode `setQwertyMode()`.

Les identifiants de choix et de groupe sont des clés servant à déverrouiller certaines fonctionnalités supplémentaires des menus :

- Un appel à `MenuItem#setCheckable()` avec un identifiant de choix ajoute une case à cocher à côté du titre de ce choix. La valeur de cette case bascule lorsque l'utilisateur sélectionne ce choix.
- Un appel à `Menu#setGroupCheckable()` avec un identifiant de groupe permet de rendre un ensemble de choix mutuellement exclusifs en leur associant des boutons radio, afin de ne pouvoir cocher qu'une seule option du groupe à un instant donné.

Enfin, il est possible de créer des sous-menus à la volée, en appelant `addSubMenu()` avec les mêmes paramètres que ceux d'`addMenu()`. Android appellera alors `onCreatePanelMenu()` en lui passant l'identifiant du choix du sous-menu, ainsi qu'une autre instance de `Menu` représentant le sous-menu lui-même. Comme avec `onCreateOptionsMenu()`, vous devez établir un chaînage avec la méthode de la superclasse, puis ajouter les choix au sous-menu. La seule restriction est que vous ne pouvez pas imbriquer sans fin les sous-menus : un menu peut avoir un sous-menu, mais ce dernier ne peut pas contenir lui-même de sous-sous-menu.

Votre activité sera prévenue d'un choix de l'utilisateur par la méthode de rappel `onOptionsItemSelected()`. Vous recevrez alors l'objet `MenuItem` correspondant à ce choix. Un motif de conception classique consiste à utiliser une instruction `switch()` avec l'identifiant du menu (`item.getItemId()`), afin d'exécuter l'action appropriée. Notez qu'`onOptionsItemSelected()` est utilisée indépendamment du fait que l'option choisie soit un choix du menu de base ou d'un sous-menu.

Menus contextuels

L'affichage d'un menu contextuel est déclenché par un touché prolongé du widget qui fournit ce menu.

Le fonctionnement des menus contextuels est quasiment identique à celui des menus d'options. Les deux différences principales concernent leur remplissage et la façon dont vous serez informé des choix effectués par l'utilisateur.

Vous devez d'abord indiquer le ou les widgets de votre activité qui disposeront de menus contextuels. Pour cela, il suffit d'appeler la méthode `registerForContextMenu()` à partir de l'activité, en lui passant en paramètre un objet `View` – le widget qui a besoin d'un menu contextuel.

Puis vous devez implémenter la méthode `onCreateContextMenu()`, qui, entre autres choses, reçoit l'objet `View` que vous aviez fourni à `registerForContextMenu()`. Si votre activité doit construire plusieurs menus contextuels, ceci permet d'identifier le menu concerné.

La méthode `onCreateContextMenu()` reçoit en paramètre le `ContextMenu` lui-même, la `View` à laquelle est associé le menu contextuel et un objet `ContextMenu.ContextMenuInfo`, qui indique l'élément de la liste qui a été touché et maintenu par l'utilisateur au cas où vous souhaiteriez personnaliser ce menu contextuel en fonction de cette information.

Il est également important de remarquer qu'`onCreateContextMenu()` est appelée à chaque fois que le menu contextuel est sollicité. À la différence d'un menu d'options (qui n'est construit qu'une seule fois par activité), les menus contextuels sont supprimés après utilisation :vous ne pouvez donc pas compter sur l'objet `ContextMenu` fourni ; il faut reconstruire le menu pour qu'il corresponde aux besoins de votre activité.

Pour être prévenu de la sélection d'un choix de menu contextuel, implémentez la méthode `onContextItemSelected()` de l'activité. Dans cette méthode de rappel, vous n'obtiendrez que l'instance du `MenuItem` qui a été choisi : si l'activité a plusieurs menus contextuels, il faut donc que les identifiants des éléments de menus soient uniques afin de pouvoir les traiter de façon appropriée. Vous pouvez également appeler la méthode `getMenuInfo()` du `MenuItem` afin d'obtenir l'objet `ContextMenu.ContextMenuInfo` que vous aviez reçu dans `onCreateContextMenu()`. Pour le reste, cette méthode de rappel se comporte comme `onOptionsItemSelected()`, que nous avons décrite dans la section précédente.

Illustration rapide

Le projet `Menus/Menus` contient une version modifiée de `Selection/List` (voir Chapitre 7) avec un menu associé. Les menus étant définis dans le code Java, le fichier de description XML n'est pas modifié et ne sera donc pas reproduit ici.

Le code Java comprend en revanche quelques nouveautés :

```
public class MenuDemo 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"};
  public static final int EIGHT_ID = Menu.FIRST+1;
  public static final int SIXTEEN_ID = Menu.FIRST+2;
```

```
public static final int TWENTY_FOUR_ID = Menu.FIRST+3;
public static final int TWO_ID = Menu.FIRST+4;
public static final int THIRTY_TWO_ID = Menu.FIRST+5;
public static final int FORTY_ID = Menu.FIRST+6;
public static final int ONE_ID = Menu.FIRST+7;
@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);
  setContentView(R.layout.main);
  setListAdapter(new ArrayAdapter<String>(this,
              android.R.layout.simple_list_item_1, items));
  selection=(TextView)findViewById(R.id.selection);

  registerForContextMenu(getListView());
}

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

@Override
public void onCreateContextMenu(ContextMenu menu, View v,
                                 ContextMenu.ContextMenuInfo menuInfo) {
  populateMenu(menu);
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
  populateMenu(menu);
  return(super.onCreateOptionsMenu(menu));
}
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  return(applyMenuChoice(item) ||
          super.onOptionsItemSelected(item));
}
@Override
public boolean onContextItemSelected(MenuItem item) {
```

```
    return(applyMenuChoice(item) ||
            super.onContextItemSelected(item));
  }

  private void populateMenu(Menu menu) {
    menu.add(Menu.NONE, ONE_ID, Menu.NONE, "1 Pixel");
    menu.add(Menu.NONE, TWO_ID, Menu.NONE, "2 Pixels");
    menu.add(Menu.NONE, EIGHT_ID, Menu.NONE, "8 Pixels");
    menu.add(Menu.NONE, SIXTEEN_ID, Menu.NONE, "16 Pixels");
    menu.add(Menu.NONE, TWENTY_FOUR_ID, Menu.NONE, "24 Pixels");
    menu.add(Menu.NONE, THIRTY_TWO_ID, Menu.NONE, "32 Pixels");
    menu.add(Menu.NONE, FORTY_ID, Menu.NONE, "40 Pixels");
  }

  private boolean applyMenuChoice(MenuItem item) {
    switch (item.getItemId()) {
      case ONE_ID:
        getListView().setDividerHeight(1);
        return(true);

      case EIGHT_ID:
        getListView().setDividerHeight(8);
        return(true);

      case SIXTEEN_ID:
        getListView().setDividerHeight(16);
        return(true);

      case TWENTY_FOUR_ID:
        getListView().setDividerHeight(24);
        return(true);

      case TWO_ID:
        getListView().setDividerHeight(2);
        return(true);

      case THIRTY_TWO_ID:
        getListView().setDividerHeight(32);
        return(true);
```

```
            case FORTY_ID:
              getListView().setDividerHeight(40);
              return(true);
          }
          return(false);
        }
      }
```

Dans `onCreate()`, nous précisons que le widget liste dispose d'un menu contextuel que nous remplissons à l'aide de notre méthode privée `populateMenu()` par l'intermédiaire d'`onCreateContextMenu()`.

Nous implémentons également la méthode de rappel `onCreateOptionsMenu()` pour signaler que l'activité dispose également d'un menu d'options rempli, lui aussi, grâce à `populateMenu()`.

Nos implémentations d'`onOptionsItemSelected()` (pour les choix dans le menu d'options) et d'`onContextItemSelected()` (pour les choix dans le menu contextuel) délèguent toutes les deux leur traitement à une méthode `applyMenuChoice()` et appellent leur superclasse respective si l'utilisateur n'a sélectionné aucun choix du menu.

`populateMenu()` ajoute sept choix de menus, chacun ayant un identifiant unique. Comme nous sommes paresseux, nous n'utiliserons pas d'icône.

Dans `applyMenuChoice()`, nous testons si l'un des choix du menu a été sélectionné, auquel cas nous fixons l'épaisseur du séparateur de la liste à la valeur correspondante. Comme le montre la Figure 11.1, l'activité initiale a le même aspect que `ListDemo`.

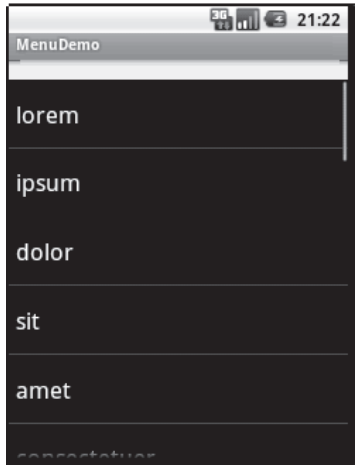
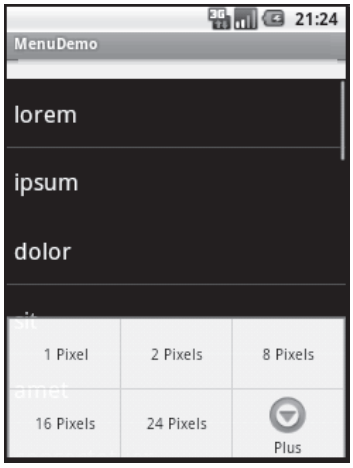


Figure 11.1
L'application `MenuDemo` *lors de son lancement.*

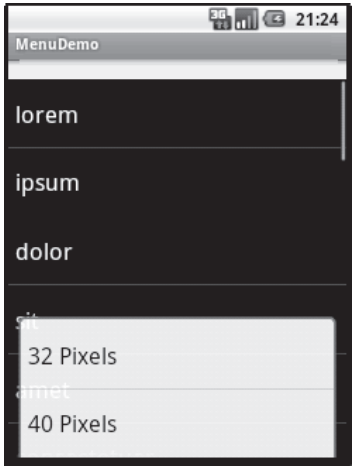
Si l'on appuie sur le bouton Menu du terminal, on obtient le menu d'options présenté à la Figure 11.2.

Figure 11.2
La même application, avec son menu d'options.



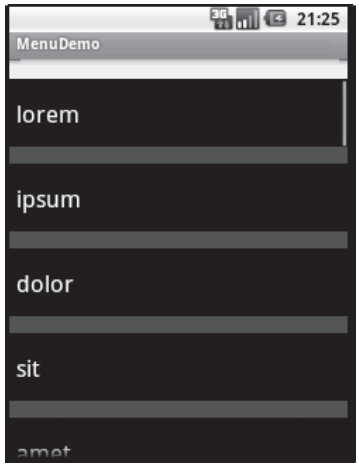
La Figure 11.3 montre que les deux choix restants du menu apparaissent lorsque l'on clique sur le bouton "Plus".

Figure 11.3
La même application, montrant les derniers choix du menu.



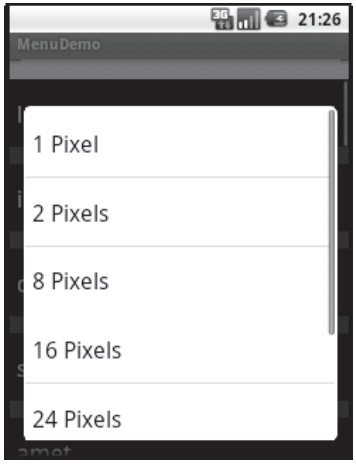
Si l'on choisit une valeur (16 pixels, par exemple), l'épaisseur des traits de séparation de la liste est modifiée, comme le montre la Figure 11.4.

Figure 11.4
La même application dans une version peu esthétique.



La Figure 11.5 montre que l'on peut faire apparaître le menu contextuel en "touchant et maintenant" un élément de la liste. Là encore, le choix d'une option modifiera l'épaisseur du trait de séparation.

Figure 11.5
La même application, avec son menu contextuel.



Encore de l'inflation

Nous avons vu au Chapitre 8 que l'on pouvait décrire les vues *via* des fichiers XML et les transformer par inflation en objets `View` au moment de l'exécution. Android permet de faire de même avec les menus, qui eux aussi peuvent être décrits dans des fichiers XML et

transformés en objets lorsqu'ils sont appelés. Cette approche permet de séparer la structure des menus de leur implémentation et facilite le développement des outils de conception de menus.

Structure XML d'un menu

Les fichiers XML de description des menus sont placés dans le répertoire `res/menu/`, à côté des autres types de ressources du projet. Comme pour les layouts, vous pouvez utiliser plusieurs fichiers de menus XML, chacun ayant son propre nom et l'extension `.xml`.

Voici, par exemple, le contenu du fichier `sample.xml`, extrait du projet `Menus/Inflation` :

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/close"
    android:title="Fermer"
    android:orderInCategory="3"
    android:icon="@drawable/eject" />
  <item android:id="@+id/no_icon"
    android:orderInCategory="2"
    android:title="Sans Icône" />
  <item android:id="@+id/disabled"
    android:orderInCategory="4"
    android:enabled="false"
    android:title="Inactif" />
  <group android:id="@+id/other_stuff"
    android:menuCategory="secondary"
    android:visible="false">
    <item android:id="@+id/later"
      android:orderInCategory="0"
      android:title="Avant-dernier" />
    <item android:id="@+id/last"
      android:orderInCategory="1"
      android:title="Dernier" />
  </group>
  <item android:id="@+id/submenu"
    android:orderInCategory="3"
    android:title="Sous-menu">
    <menu>
      <item android:id="@+id/non_ghost"
        android:title="Visible"
        android:visible="true"
        android:alphabeticShortcut="v" />
      <item android:id="@+id/ghost"
```

```
            android:title="Fantôme"
            android:visible="false"
            android:alphabeticShortcut="f" />
        </menu>
      </item>
    </menu>
```

Voici quelques remarques à propos des définitions des menus en XML :

- L'élément racine doit s'appeler `menu`.
- Les éléments fils de `menu` sont des objets `item` et `group`, ces derniers représentant une collection d'objets `item` pouvant être manipulés comme un groupe.
- Les sous-menus sont déclarés en ajoutant un élément `menu` comme fils d'un élément `item` et en utilisant ce nouvel élément pour décrire le contenu du sous-menu.
- Pour détecter qu'un choix a été sélectionné ou faire référence à un choix ou à un groupe à partir de Java, n'oubliez pas d'utiliser l'attribut `android:id`, exactement comme pour les fichiers XML des vues.

Options des menus et XML

Les éléments `item` et `group` peuvent contenir plusieurs attributs correspondant aux méthodes de `Menu` et `MenuItem` :

- **Titre.** Le titre d'un choix de menu est fourni par l'attribut `android:title` d'un élément `item`. Ce titre peut être une chaîne littérale ou une référence vers une ressource chaîne (`@string/truc`, par exemple).
- **Icône.** Les choix de menus peuvent posséder des icônes qui sont fournies sous la forme d'une référence vers une ressource `Drawable` (`@drawable/eject`, par exemple) *via* l'attribut `android:icon` d'un élément `item`.
- **Ordre.** Par défaut, l'ordre des choix du menu est déterminé par celui de leur apparition dans le fichier XML. Vous pouvez modifier cet ordre à l'aide de l'attribut `android:orderInCategory` de l'élément `item`. Sa valeur est un indice commençant à 0, qui précise son ordre d'apparition dans la catégorie courante. Les éléments `group` peuvent utiliser l'attribut `android:menuCategory` pour indiquer une catégorie différente de celle par défaut pour leurs éléments `item`.

  Cependant, il est généralement plus simple de placer les éléments dans le bon ordre dans le fichier XML.
- **Activation.** Les `item` et les `group` peuvent être rendus actifs ou inactifs dans le fichier XML à l'aide de leur attribut `android:enabled`. Par défaut, ils sont considérés comme actifs. Les objets `item` et `group` inactifs apparaissent dans le menu mais ne peuvent pas

être sélectionnés. Vous pouvez modifier l'état d'un `item` en cours d'exécution grâce à la méthode `setEnabled()` de `MenuItem` et celui d'un `group` *via* la méthode `setGroupEnabled()` de `Menu`.

- **Visibilité.** Les objets `item` et `group` peuvent être visibles ou non, selon la valeur de leur attribut `android:visible`. Par défaut, ils sont visibles ; les objets `item` et `group` invisibles n'apparaissent pas dans le menu. Vous pouvez modifier cet état en cours d'exécution en appelant la méthode `setVisible()` d'un `MenuItem` ou la méthode `setGroupVisible()` d'un `Menu`.

  Dans le fichier XML présenté plus haut, le groupe `other_stuff` est invisible au départ. Si nous le rendons visible dans le code Java, les deux choix du groupe apparaîtront comme par magie.

- **Raccourci.** Les `item` d'un menu peuvent avoir des raccourcis – en appuyant sur une seule lettre (`android:alphabeticShortcut`) ou un chiffre (`android:numericShortcut`), vous pouvez sélectionner cet `item` sans utiliser l'écran tactile, le pad ou le curseur pour naviguer dans le menu.

Création d'un menu par inflation

L'utilisation d'un menu défini dans un fichier XML est relativement simple puisqu'il suffit de créer un `MenuInflater` et de lui demander de créer l'objet `Menu` :

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  theMenu=menu;
  new MenuInflater(getApplication())
                              .inflate(R.menu.sample, menu);
  return(super.onCreateOptionsMenu(menu));
}
```

12

Polices de caractères

Inévitablement, un développeur d'applications doit répondre à la question : "Comment changer cette police de caractères ?" La réponse dépend des polices fournies avec la plateforme, de la possibilité d'en ajouter d'autres et du moyen de les appliquer au widget ou à la partie de l'application concernée. Android n'y fait pas exception car il est fourni avec quelques polices et permet d'en ajouter d'autres. Toutefois, comme avec tout nouvel environnement, il faut savoir gérer ses spécificités.

Sachez apprécier ce que vous avez

Nativement, Android connaît trois polices sous les noms raccourcis `"sans"`, `"serif"` et `"monospace"`. Elles forment la série Droid, créée par Ascender[1] pour l'Open Handset Alliance.

Pour les utiliser, il suffit de les désigner dans le fichier de description XML. Le layout suivant, par exemple, est extrait du projet `Fonts/FontSampler` :

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout
```

1. **http://www.ascendercorp.com/oha.html**.

```
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="sans : "
    android:layout_marginRight="4px"
    android:textSize="20sp"
  />
  <TextView
    android:id="@+id/sans"
    android:text="Bonjour !"
    android:typeface="sans"
    android:textSize="20sp"
  />
</TableRow>
<TableRow>
  <TextView
    android:text="serif : "
    android:layout_marginRight="4px"
    android:textSize="20sp"
  />
  <TextView
    android:id="@+id/serif"
    android:text="Bonjour !"
    android:typeface="serif"
    android:textSize="20sp"
  />
</TableRow>
<TableRow>
  <TextView
    android:text="monospace : "
    android:layout_marginRight="4px"
    android:textSize="20sp"
  />
  <TextView
    android:id="@+id/monospace"
    android:text="Bonjour !"
    android:typeface="monospace"
    android:textSize="20sp"
  />
</TableRow>
<TableRow>
  <TextView
    android:text="Custom : "
    android:layout_marginRight="4px"
```

```
        android:textSize="20sp"
      />
      <TextView
        android:id="@+id/custom"
        android:text="Bonjour !"
        android:textSize="20sp"
      />
    </TableRow>
  </TableLayout>
```

Cette description construit un tableau montrant les noms courts de quatre polices. Vous remarquerez que les trois premiers utilisent l'attribut `android:typeface`, dont la valeur est l'une des trois polices prédéfinies (`"sans"`, par exemple).

Polices supplémentaires

Ces trois polices sont parfaites. Cependant, un designer, un chef de projet ou un client peut très bien vouloir en utiliser une autre. Vous pouvez également choisir d'utiliser une police spéciale, comme "dingbats", à la place d'images PNG.

Le moyen le plus simple d'y parvenir consiste à fournir la ou les polices concernées avec votre application. Pour ce faire, il suffit de créer un répertoire `assets/` dans l'arborescence du projet et d'y placer les polices TrueType (TTF). Vous pourriez, par exemple, créer un répertoire `assets/polices/` et y placer les fichiers TTF.

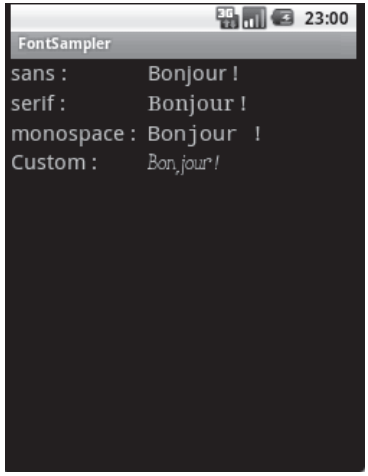
Il faut ensuite demander aux widgets d'utiliser cette police mais, malheureusement, vous ne pouvez pas le faire dans le fichier de description XML car il ne connaît pas celles que vous avez pu placer dans les assets de l'application. Cette modification doit donc avoir lieu dans le code Java :

```
public class FontSampler extends Activity {
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);

    TextView tv=(TextView)findViewById(R.id.custom);
    Typeface face=Typeface.createFromAsset(getAssets(),
                                           "polices/HandmadeTypewriter.ttf");

    tv.setTypeface(face);
  }
}
```

Ici, on récupère le `TextView` pour notre exemple "Custom" et l'on crée un objet `Typeface` en appelant la méthode statique `createFromAsset()`, qui prend en paramètre l'`AssetManager` de l'application (obtenu *via* `getAssets()`) et un chemin relatif de la police présent dans le répertoire `assets/`.

Ensuite, il suffit d'indiquer au `TextView` d'utiliser cette police en appelant sa méthode `setTypeface()` avec le `Typeface` que l'on vient de créer. Ici, on utilise la police Handmade Typewriter[1] (voir Figure 12.1).

Figure 12.1
L'application `FontSampler`.



Android semble ne pas apprécier toutes les fontes TrueType. Lorsqu'il n'aime pas une fonte extérieure, Android lui substitue sans mot dire la police Droid Sans ("`sans`*") au lieu de lever une exception. Par conséquent, si vous essayez d'utiliser une police et qu'elle semble ne pas fonctionner, c'est sûrement parce qu'elle est incompatible avec Android, pour une raison ou pour une autre.*

En outre, vous avez intérêt à mettre en minuscules les noms des fichiers de polices, afin de respecter les conventions utilisées par vos autres ressources.

Android 1.6 a ajouté la possibilité de créer avec la méthode `createFromFile()` *des objets* `Typeface` *à partir de fichiers TrueType situés sur le système de fichiers – sur la carte SD de l'utilisateur, par exemple.*

1. **http://moorstation.org/typoasis/designers/klein07/text01/handmade.htm**.

Le problème des glyphes

Les polices TrueType peuvent être assez lourdes, notamment lorsqu'elles fournissent un sous-ensemble important des caractères Unicode. La police Handmade Typewriter que nous avons utilisée ci-dessus, par exemple, pèse plus de 70 Ko et les polices libres DejaVu peuvent atteindre 500 Ko chacune. Même compressées, elles augmentent donc significativement la taille de votre application : ne vous laissez pas envahir par les polices supplémentaires sous peine de produire une application qui prendra trop de place sur les téléphones des utilisateurs.

Inversement, il faut savoir que les polices ne contiennent pas nécessairement tous les glyphes dont vous avez besoin. Par exemple, la classe `TextView` d'Android sait créer une ellipse d'un texte, en le tronquant et en ajoutant un caractère d'ellipse si le texte ne peut pas tenir dans l'espace disponible. Ce comportement peut être obtenu *via* l'attribut `android:ellipsize`, par exemple. Ceci fonctionne très bien, au moins pour les textes d'une seule ligne.

L'ellipse utilisée par Android n'est pas une simple suite de trois points, mais un véritable caractère "ellipse", où les trois points sont contenus dans un unique glyphe. Par conséquent, les polices que vous utilisez auront besoin de ce glyphe pour utiliser la fonctionnalité fournie par `android:ellipsize`.

En outre, Android complète les chaînes qui s'affichent à l'écran pour que leur longueur (en nombre de caractères) soit identique à celles qu'elles avaient avant leur mise en ellipse. Pour que cela puisse fonctionner, Android remplace un seul caractère par l'ellipse et tous ceux qui seront supprimés par le caractère Unicode "ZERO WIDTH NO-BREAK SPACE" (U+FEFF). Ceci signifie que les caractères "supplémentaires" après l'ellipse font partie de la chaîne, bien qu'ils n'apparaissent pas à l'écran.

Les polices que vous utilisez avec `TextView` et l'attribut `android:ellipsize` doivent donc également disposer de ce caractère Unicode, ce qui n'est pas toujours le cas. En son absence, l'affichage de la chaîne écourtée à l'écran présentera des caractères curieux (des X à la fin de la ligne, par exemple).

Bien entendu, le déploiement international d'Android signifie également qu'une police doit pouvoir gérer n'importe quelle langue choisie par l'utilisateur, avec ses caractères particuliers (le é du français, par exemple).

Par conséquent, bien qu'il soit tout à fait possible d'utiliser des polices personnalisées avec Android, cela induit de nombreux problèmes potentiels et vous devez donc soigneusement peser le pour et le contre.

13

Intégrer le navigateur de WebKit

Les autres kits de développement graphiques permettent d'utiliser HTML pour présenter les informations, que ce soit *via* des minimoteurs HTML (wxWidgets en Java/Swing, par exemple) ou par l'intégration d'Internet Explorer dans les applications .NET. Android fait de même en vous permettant d'intégrer un navigateur web dans vos activités afin d'afficher du HTML ou de naviguer sur le Web. Ce navigateur repose sur WebKit, le moteur de Safari (le navigateur d'Apple).

Ce navigateur est suffisamment complexe pour disposer de son propre paquetage Java (`android.webkit`). La simplicité d'utilisation du widget `WebView`, quant à elle, dépend de vos exigences.

Un navigateur, et en vitesse !

Pour les opérations simples, `WebView` n'est pas très différent des autres widgets d'Android – on le fait surgir dans un layout, on lui indique dans le code Java l'URL vers laquelle il doit naviguer et c'est fini.

Le layout du projet `WebKit/Browser1`, par exemple, ne contient qu'un `WebView` :

```
<?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"
  >
  <WebView android:id="@+id/webkit"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
  />
</LinearLayout>
```

Comme avec tous les autres widgets, vous devez lui indiquer comment remplir l'espace (ici, il occupe tout l'espace restant).

Le code Java est tout aussi simple :

```
package com.commonsware.android.browser1;

import android.app.Activity;
import android.os.Bundle;
import android.webkit.WebView;

public class BrowserDemo1 extends Activity {
  WebView browser;

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

    browser.loadUrl("http://www.pearson.fr");
  }
}
```

La seule partie nouvelle de cette version d'`onCreate()` est l'appel de la méthode `loadUrl()` du widget `WebView` afin de charger une page web.

Nous devons également modifier le fichier `AndroidManifest.xml`, afin de demander la permission d'accéder à Internet ; sinon le navigateur refusera de charger les pages :

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.browser1">
  <uses-permission android:name="android.permission.INTERNET" />
  <application android:icon="@drawable/cw">
    <activity android:name=".BrowserDemo1" android:label="BrowserDemo1">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Comme le montre la Figure 13.1, l'activité obtenue ressemble à un navigateur web sans barre de défilement.

Figure 13.1
L'application `Browser1`.

Comme avec le navigateur classique d'Android, vous pouvez faire défiler la page en la tirant avec le doigt et le pad directionnel permet de vous positionner successivement sur tous ses éléments actifs.

Il manque toutefois certaines parties pour en faire un véritable navigateur : une barre de navigation, notamment.

Vous pourriez être tenté de remplacer l'URL de ce code source par une autre, comme celle de la page d'accueil de Google ou toute autre page utilisant JavaScript. Cependant, par défaut, un widget `WebView` désactive JavaScript : pour l'activer, vous devez appeler sa méthode `getSettings().setJavaScriptEnabled(true);`.

Chargement immédiat

Il existe deux moyens de faire entrer du contenu dans un widget `WebView`. Le premier, comme nous l'avons vu, consiste à indiquer une URL au navigateur et à lui faire afficher cette page *via* `loadUrl()`. Le navigateur accédera à Internet par les moyens mis à disposition du terminal à cet instant précis (Wifi, réseau cellulaire, partage de données par Bluetooth, pigeons voyageurs bien entraînés, etc.).

L'autre solution consiste à utiliser `loadData()` en lui fournissant le code HTML que l'on veut afficher dans le navigateur. Cette dernière méthode est notamment utile pour :

- afficher un manuel installé sous forme de fichier avec l'application ;
- afficher des extraits HTML récupérés par un autre traitement – la description d'une entrée d'un flux Atom, par exemple ;
- produire une interface utilisateur entièrement en HTML au lieu d'utiliser les widgets Android.

Il existe deux variantes de `loadData()`. La plus simple permet de fournir sous forme de chaînes un contenu, son type MIME et l'encodage utilisé. Pour un document HTML classique, le type MIME sera généralement `text/html` et l'encodage, UTF-8.

Si vous remplacez l'appel de `loadUrl()` par le code suivant dans l'exemple précédent, vous obtiendrez le résultat présenté à la Figure 13.2.

```
browser.loadData("<html><body>Bonjour !</body></html>",
                 "text/html", "UTF-8");
```

Figure 13.2
L'application `Browser2`.

Le code complet de cette application est disponible dans le projet `WebKit/Browser2`.

Navigation au long cours

Comme on l'a déjà mentionné, le widget `WebView` ne propose pas de barre de navigation, ce qui permet de l'utiliser à des endroits où cette barre serait inutile et consommerait inutilement de l'espace à l'écran. Ceci étant dit, il est tout à fait possible de lui ajouter des fonctionnalités de navigation, à condition de fournir l'interface graphique nécessaire.

`WebView` dispose notamment des méthodes suivantes :

- `reload()` permet de recharger la page courante.
- `goBack()` permet de revenir un pas en arrière dans l'historique du navigateur, tandis que `canGoBack()` teste s'il existe un historique passé.
- `goForward()` permet d'avancer d'un pas dans l'historique du navigateur, tandis que `canGoForward()` teste s'il existe un historique futur.
- `goBackOrForward()` permet de reculer ou d'avancer dans l'historique en fonction de son paramètre. Une valeur négative représente le nombre de pas vers l'arrière, une valeur positive, le nombre de pas vers l'avant.
- `canGoBackOrForward()` teste si le navigateur peut reculer ou avancer du nombre de pas indiqué dans l'historique (en suivant la même convention positif/négatif que pour `goBackOrForward()`).
- `clearCache()` vide le cache du navigateur et `clearHistory()` nettoie son historique.

Amuser le client

Si vous utilisez un widget `WebView` pour créer une interface utilisateur locale (et non pour naviguer sur le Web), vous aurez besoin de le contrôler à certains moments, notamment lorsque les utilisateurs cliqueront sur des liens. Vous devrez vous assurer que ces liens sont correctement gérés, soit en rechargeant votre propre contenu dans le `WebView` (en soumettant un `Intent` à Android pour qu'il ouvre l'URL dans un vrai navigateur), soit par d'autres moyens (voir Chapitre 18).

La méthode `setWebViewClient()`, qui prend en paramètre une instance d'une implémentation de `WebViewClient`, permet de placer un *hook* dans l'activité `WebView`. L'objet fourni sera alors prévenu d'un grand nombre d'activités, allant de la récupération d'une partie d'une page (`onPageStarted()`, etc.) à la nécessité pour l'application de prendre en compte un événement utilisateur ou circonstanciel (`onTooManyRedirects()`, `onReceivedHttpAuthRequest()`, etc.).

Un hook classique est `shouldOverrideUrlLoading()`, qui prend en paramètre une URL et l'objet `WebView` lui-même et qui doit renvoyer `true` si vous traitez la requête ou `false` si

vous préférez utiliser le traitement par défaut (qui consiste à récupérer la page web désignée par l'URL). Pour une application de lecture de flux RSS, par exemple, le lecteur ne comprendra sûrement pas un navigateur web complet, avec toutes les options de navigation possibles ; si l'utilisateur clique sur une URL, vous utiliserez donc probablement un `Intent` pour demander à Android de charger cette page dans un vrai navigateur. Cependant, si vous avez inséré une URL "bidon" dans le code HTML, représentant un lien vers un contenu fourni par une activité, vous pouvez modifier vous-même le `WebView`.

Modifions notre premier exemple pour qu'il devienne l'équivalent de la première application de ce livre : un programme qui affiche la date et l'heure courantes lorsque l'on clique sur un bouton. Le code Java de ce projet `WebKit/Browser3` est le suivant :

```
public class BrowserDemo3 extends Activity {
  WebView browser;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    browser=(WebView)findViewById(R.id.webkit);
    browser.setWebViewClient(new Callback());

    loadTime();
  }

  void loadTime() {
    String page="<html><body><a href=\"clock\">"
            +new Date().toString()
            +"</a></body></html>";

            browser.loadDataWithBaseURL("x-data://base", page,
                                        "text/html", "UTF-8",
                                        null);
  }
  private class Callback extends WebViewClient {
    public boolean shouldOverrideUrlLoading(WebView view, String url) {
      loadTime();

      return(true);
    }
  }
}
```

On charge une simple page web dans le navigateur (*via* la méthode `loadTime()`). Cette page contient la date et l'heure courantes sous la forme d'un lien vers l'URL `/clock`. On attache également une instance d'une sous-classe de `WebViewClient` en fournissant notre implémentation de `shouldOverrideUrlLoading()`. Ici, peu importe l'URL : nous voulons simplement recharger le `WebView` avec `loadTime()`.

Le résultat de l'exécution de cette activité est présenté à la Figure 13.3.

Figure 13.3
L'application `Browser3`.

En sélectionnant le lien et en cliquant sur le bouton central du pad, ce lien sera considéré comme "cliqué", ce qui aura pour effet de reconstruire la page avec une nouvelle date/heure.

Réglages, préférences et options

Votre navigateur web favori contient un menu "Réglages", "Préférences" ou "Options". Ajouté aux contrôles de la barre d'outils, ce menu vous permet d'adapter de nombreux aspects du comportement du navigateur, allant des polices qu'il utilise à la gestion de JavaScript.

De la même façon, vous pouvez modifier les réglages d'un widget `WebView` en fonction de vos besoins, *via* l'instance de `WebSettings` qui est renvoyée par la méthode `getSettings()` du widget.

Beaucoup d'options de `WebSettings` sont ainsi à votre disposition. Bien que la plupart semblent assez ésotériques (`setFantasyFontFamily()`, par exemple), quelques-unes peuvent vous être plus utiles :

- Les méthodes `setDefaultFontSize()` (pour une taille en points) ou `setTextSize()` (qui utilise des constantes comme `LARGER` et `SMALLEST` pour exprimer une taille relative) permettent de modifier la taille de la police.

- `setJavaScriptEnabled()` et `setJavaScriptCanOpenWindowsAutomatically()` permettent, respectivement, de désactiver totalement JavaScript et de l'empêcher d'ouvrir des fenêtres popup.
- `setUserAgent()` permet de contrôler le rendu d'un site web. Si son paramètre vaut 0, le `WebView` passera au site une chaîne `user-agent` indiquant qu'il s'agit d'un navigateur mobile alors que, s'il vaut 1, le widget enverra une chaîne indiquant qu'il s'agit d'un navigateur qui s'exécute sur une machine de bureau.

Les modifications que vous apportez ne sont pas persistantes : vous devez donc les stocker quelque part (*via* le moteur de préférences d'Android que nous présenterons au Chapitre 21, par exemple) si vous autorisez vos utilisateurs à modifier ces réglages au lieu de les coder en dur dans votre application.

14

Affichage de messages surgissants

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 31. 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 une instance `Toast` à l'aide du 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 de 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="Déclencher 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 en profiter pour 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.

Figure 14.1
L'application `MessageDemo` *après avoir cliqué sur le bouton Déclencher une alerte.*

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 14.2
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, ce qui implique invariablement d'utiliser un thread. 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.

Cette "interaction propre" est cruciale car l'interface graphique ne peut être modifiée que par son thread et non par un thread en arrière-plan. Ceci signifie généralement qu'il devra donc exister une certaine coordination entre les threads en arrière-plan, qui effectuent le traitement, et le thread de l'interface, qui affiche le résultat de ce traitement.

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 contenant des identifiants et des paramètres de messages. 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, 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, exprimé 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` pour demander ici que la barre de progression soit 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 puis nous sortons du gestionnaire de message.

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 la position de un 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`.

Notez que, bien que le code de cet exemple s'arrange pour mettre à jour la barre de progression dans le thread de l'interface graphique, ce n'est pas nécessaire. À partir d'Android 1.5, en effet, `ProgressBar` est *thread safe*, ce qui signifie que vous pouvez la modifier à partir de n'importe quel thread et qu'elle gérera les détails de la modification de l'interface graphique dans le thread de celle-ci.

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()`. Celle-ci 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 supprimera 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 13 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 l'implémentation de l'interface `Runnable`, par exemple, 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 obligatoirement 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 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 de 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`.

L'exécution de cette activité construit la liste de mots en quelques secondes et en temps réel, puis lance un toast pour signaler la fin du traitement (voir Figure 15.2).

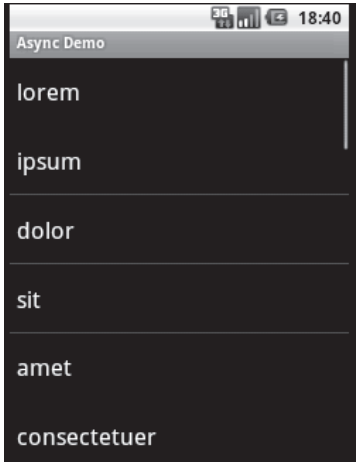


Figure 15.2
L'activité `AsyncDemo` *en cours de construction de la liste des mots.*

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 un 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, nous pourrions en profiter pour supprimer une `ProgressBar` ou 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()`.

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. Notamment :

- 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 Chapitre 16 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 – le terminal peut perdre sa connectivité pendant que vous récupérez des informations à partir d'Internet, par exemple. La meilleure solution consiste alors à prévenir l'utilisateur du problème *via* une `Notification` (voir Chapitre 31) puis à mettre fin au thread en arrière-plan.

16

Gestion des événements du cycle de vie d'une activité

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

Par conséquent, votre activité risque de se faire tuer parce que d'autres activités ont lieu et que le système a besoin de la mémoire qu'elle occupe. Il faut considérer tout cela comme l'équivalent Android du "cercle de la vie" – votre activité meurt afin que d'autres puissent vivre, etc.

Vous ne pouvez donc pas supposer que votre application s'exécutera jusqu'à son terme, que ce soit de votre point de vue ou de celui de l'utilisateur.

C'est l'un des exemples – peut-être le plus important – de l'impact du cycle de vie d'une activité sur le déroulement de vos propres applications.

Dans ce chapitre, nous présenterons les différents états qui composent le cycle de vie d'une activité et la façon de les gérer correctement.

L'activité de Schroedinger

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

- **Active.** L'activité a été lancée par l'utilisateur, elle s'exécute au premier plan. C'est à cet état que l'on pense quand on évoque le fonctionnement d'une activité.
- **En pause.** L'activité a été lancée par l'utilisateur, elle s'exécute et elle est visible, mais une notification ou un autre événement occupe une partie de l'écran. Pendant ce temps, l'utilisateur voit l'activité mais peut ne pas être capable d'interagir avec elle. Lorsqu'un appel téléphonique est reçu, l'utilisateur a l'opportunité de prendre cet appel ou de l'ignorer, par exemple.
- **Stoppée.** L'activité a été lancée par l'utilisateur, elle s'exécute mais est cachée par d'autres activités qui ont été lancées ou vers lesquelles le système a basculé. Votre application ne pourra rien présenter d'intéressant à l'utilisateur directement : elle ne peut passer que par une notification (voir Chapitre 31).
- **Morte.** L'activité n'a jamais été lancée (le téléphone vient d'être réinitialisé, par exemple) ou elle a été tuée, éventuellement à cause d'un manque de mémoire.

Vie et mort d'une activité

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

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

onCreate() et *onDestroy()*

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

Cette méthode sera appelée dans les trois cas suivants :

- Lorsque l'activité est lancée pour la première fois (après le redémarrage du système, par exemple), `onCreate()` est appelée avec le paramètre `null`.
- Si l'activité s'est exécutée, puis qu'elle a été tuée, `onCreate()` sera appelée avec, pour paramètre, le `Bundle` obtenu par `onSaveInstanceState()` (voir plus loin).
- Si l'activité s'est exécutée et que vous l'ayez configurée pour qu'elle utilise des ressources différentes en fonction des états du terminal (mode portrait ou mode paysage, par exemple), elle sera recréée et `onCreate()` sera donc appelée. Les ressources seront présentées au Chapitre 20.

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

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

onStart(), *onRestart()* et *onStop()*

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

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

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

onPause() et *onResume()*

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

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

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

L'état de grâce

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

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

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

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

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

17

Création de filtres d'intentions

Pour l'instant, nous ne nous sommes intéressés qu'aux activités ouvertes directement par l'utilisateur à partir du lanceur du terminal, ce qui est, évidemment, le moyen le plus évident de lancer une activité et de la rendre disponible à l'utilisateur. Dans la plupart des cas, c'est de cette façon que l'utilisateur commencera à utiliser votre application.

Cependant, le système Android repose sur un grand nombre de composants étroitement liés. Ce que vous pouvez obtenir dans une interface graphique *via* des boîtes de dialogue, des fenêtres filles, etc. est généralement traité par des activités indépendantes. Bien que l'une d'elles puisse être "spécifique" puisqu'elle apparaît dans le lanceur, les autres doivent toutes être accessibles... d'une façon ou d'une autre.

Elles le sont grâce aux intentions.

Une intention est essentiellement un message que l'on passe à Android pour lui dire "je veux que tu fasses... quelque chose". Ce "quelque chose" dépend de la situation – parfois,

on sait parfaitement de quoi il s'agit (ouvrir l'une de nos autres activités, par exemple) mais, d'autres fois, on ne le sait pas.

Dans l'absolu, Android ne se consacre qu'aux intentions et à leurs récepteurs. Maintenant que nous avons vu comment créer des activités, plongeons-nous dans les intentions afin de pouvoir créer des applications plus complexes tout en étant de "bons citoyens d'Android".

Quelle est votre intention ?

Lorsque sir Tim Berners-Lee a conçu le protocole de transfert hypertexte HTTP, il a défini un ensemble de verbes et d'adresses sous la forme d'URL. Une adresse désigne une ressource : une page web, une image ou un programme qui s'exécute sur un serveur, par exemple. Un verbe précise l'action qui doit s'appliquer à cette adresse : `GET` pour la récupérer, `POST` pour lui envoyer des données de formulaire afin qu'elle les traite, etc.

Les intentions sont similaires car elles représentent une action et un contexte. Bien qu'elles permettent de définir plus d'actions et de composants de contexte qu'il n'y a de verbes et de ressources HTTP, le concept est le même.

Tout comme un navigateur web sait comment traiter une paire verbe + URL, Android sait comment trouver les activités ou les autres applications qui sauront gérer une intention donnée.

Composantes des intentions

Les deux parties les plus importantes d'une intention sont l'action et ce qu'Android appelle les "données". Elles sont quasiment analogues aux verbes et aux URL de HTTP – l'action est le verbe et les "données" sont une `Uri` comme `content://contacts/people/1`, représentant un contact dans la base de données des contacts. Les actions sont des constantes, comme `ACTION_VIEW` (pour afficher la ressource), `ACTION_EDIT` (pour la modifier) ou `ACTION_PICK` (pour choisir un élément disponible dans une `Uri` représentant une collection, comme `content://contacts/people`).

Si vous créez une intention combinant `ACTION_VIEW` avec l'`Uri content://contacts/people/1` et que vous la passez à Android, ce dernier saura comment trouver et ouvrir une activité capable d'afficher cette ressource.

Outre l'action et l'`Uri` des "données", vous pouvez placer d'autres critères dans une intention (qui est représentée par un objet `Intent`) :

- **Une catégorie.** Votre activité "principale" appartient à la catégorie `LAUNCHER`, pour indiquer qu'elle apparaît dans le menu du lanceur. Les autres activités appartiendront probablement aux catégories `DEFAULT` ou `ALTERNATIVE`.

- **Un type MIME** indiquant le type de ressource sur laquelle vous voulez travailler si vous ne connaissez pas une `Uri` collection.
- **Un composant**, c'est-à-dire la classe de l'activité supposée recevoir cette intention. Cette utilisation des composants évite d'avoir besoin des autres propriétés de l'intention, mais elle rend cette dernière plus fragile car elle suppose des implémentations spécifiques.
- **Des "Extras"**, c'est-à-dire un `Bundle` d'autres informations que vous voulez passer au récepteur en même temps que l'intention et dont ce dernier pourra tirer parti. Les informations utilisables par un récepteur donné dépendent du récepteur et sont (heureusement) bien documentées.

La documentation d'Android consacrée à la classe `Intent` contient les listes des actions et des catégories standard.

Routage des intentions

Comme on l'a mentionné précédemment, si le composant cible a été précisé dans l'intention, Android n'aura aucun doute sur sa destination – il lancera l'activité en question. Ce mécanisme peut convenir si l'intention cible se trouve dans votre application, mais n'est vraiment pas recommandé pour envoyer des intentions à d'autres applications car les noms des composants sont globalement considérés comme privés à l'application et peuvent donc être modifiés. Il est donc préférable d'utiliser les modèles d'`Uri` et les types MIME pour identifier les services auxquels vous souhaitez accéder.

Si vous ne précisez pas de composant cible, Android devra trouver les activités (ou les autres récepteurs d'intentions) éligibles pour cette intention. Vous aurez remarqué que nous avons mis "activités" au pluriel car une activité peut très bien se résoudre en plusieurs activités. Cette approche du routage est préférable au *routage implicite*.

Essentiellement, trois conditions doivent être vérifiées pour qu'une activité soit éligible pour une intention donnée :

- L'activité doit supporter l'action indiquée.
- L'activité doit supporter le type MIME indiqué (s'il a été fourni).
- L'activité doit supporter toutes les catégories nommées dans l'intention.

La conclusion est que vous avez intérêt à ce que vos intentions soient suffisamment spécifiques pour trouver le ou les bons récepteurs, mais pas plus.

Tout ceci deviendra plus clair à mesure que nous étudierons quelques exemples.

Déclarer vos intentions

Tous les composants Android qui souhaitent être prévenus par des intentions doivent déclarer des filtres d'intention afin qu'Android sache quelles intentions doivent aller vers quel composant. Pour ce faire, vous devez ajouter des éléments `intent-filter` au fichier `AndroidManifest.xml`.

Le script de création des applications Android (`activityCreator` ou son équivalent IDE) fournit des filtres d'intention à tous les projets. Ces déclarations sont de la forme :

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.commonsware.android.skeleton">
    <application>
        <activity android:name=".Now" android:label="Now">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Notez la présence de l'élément `intent-filter` sous l'élément `activity`. Il annonce que :

- Cette activité est l'activité principale de cette application.
- Elle appartient à la catégorie `LAUNCHER`, ce qui signifie qu'elle aura une icône dans le menu principal d'Android.

Cette activité étant l'activité principale de l'application, Android sait qu'elle est le composant qu'il devra lancer lorsqu'un utilisateur choisit cette application à partir du menu principal.

Vous pouvez bien sûr indiquer plusieurs actions ou catégories dans vos filtres d'intention afin de préciser que le composant associé (l'activité) gère plusieurs sortes d'intentions différentes.

Il est fort probable que vous voudrez également que vos activités secondaires (non `MAIN`) précisent le type MIME des données qu'elles manipulent. Ainsi, si une intention est destinée à ce type MIME – directement ou indirectement *via* une `Uri` référençant une ressource de ce type –, Android saura que le composant sait gérer ces données.

Vous pourriez, par exemple, déclarer une activité de la façon suivante :

```
<activity android:name=".TourViewActivity">
  <intent-filter>
    <action android:name="android.intent.action.VIEW" />
```

```
      <category android:name="android.intent.category.DEFAULT" />
      <data android:mimeType="vnd.android.cursor.item/vnd.commonsware.tour" />
    </intent-filter>
  </activity>
```

Celle-ci sera alors lancée par une intention demandant l'affichage d'une `Uri` représentant un contenu `vnd.android.cursor.item/vnd.commonsware.tour`. Cette intention pourrait provenir d'une autre activité de la même application (l'activité `MAIN`, par exemple) ou d'une autre application qui connaît une `Uri` que cette activité peut gérer.

Récepteurs d'intention

Dans les exemples que nous venons de voir, les filtres d'intention étaient configurés sur des activités. Cependant, lier les intentions à des activités n'est parfois pas exactement ce dont on a besoin :

- Certains événements système peuvent nous obliger à déclencher une opération dans un service plutôt qu'une activité.
- Certains événements peuvent devoir lancer des activités différentes en fonction des circonstances, où le critère repose non pas uniquement sur l'intention elle-même, mais sur un autre état (si l'on obtient l'intention *X* et que la base de données contienne *Y*, on lance l'activité *M* ; si la base ne contient pas *Y*, on lance l'activité *N,* par exemple).

Dans ces situations, Android offre un récepteur d'intention défini comme une classe qui implémente l'interface `BroadcastReceiver`. Les récepteurs d'intention sont des objets conçus pour recevoir des intentions – notamment celles qui sont diffusées – et pour effectuer une action impliquant généralement le lancement d'autres intentions pour déclencher une opération dans une activité, un service ou un autre composant.

L'interface `BroadcastReceiver` ne possède qu'une seule méthode `onReceive()` que les récepteurs d'intention doivent donc implémenter pour y effectuer les traitements qu'ils souhaitent en cas de réception d'une intention. Pour déclarer un récepteur d'intention, il suffit d'ajouter un élément `receiver` au fichier `AndroidManifest.xml` :

```
<receiver android:name=".MaClasseReceptriceDIntention/>
```

Un récepteur d'intention ne vit que le temps de traiter `onReceive()` – lorsque cette méthode se termine, l'instance est susceptible d'être supprimée par le ramasse-miettes et ne sera pas réutilisée. Ceci signifie donc que les fonctionnalités de ces récepteurs sont un peu limitées, essentiellement pour éviter l'appel de fonctions de rappel. Ils ne peuvent notamment pas être liés à un service ni ouvrir une boîte de dialogue.

La seule exception est lorsque le `BroadcastReceiver` est implémenté sur un composant qui a une durée de vie assez longue, comme une activité ou un service : dans ce cas, le récepteur vivra aussi longtemps que son "hôte" (jusqu'à ce que l'activité soit stoppée, par exemple). Cependant, dans cette situation, vous ne pouvez pas déclarer le récepteur dans `AndroidManifest.xml` : il faut appeler `registerReceiver()` dans la méthode `onResume()` de l'activité pour annoncer son intérêt pour une intention, puis appeler `unregisterReceiver()` dans sa méthode `onPause()` lorsque vous n'avez plus besoin de ces intentions.

Attention à la pause

Il y a un petit problème lorsque l'on utilise des objets `Intent` pour transmettre des messages : cela ne fonctionne que lorsque le récepteur est actif. Voici ce que précise la documentation de `BroadcastReceiver` à ce sujet :

Si vous enregistrez un récepteur dans votre implémentation d'`Activity.onResume()`, il faut le désinscrire dans `Activity.onPause()` (vous ne recevrez pas d'intention pendant la pause et cela évite une surcharge inutile du système). N'effectuez pas cette désinscription dans `Activity.onSaveInstanceState()`, car cette méthode n'est pas appelée lorsque l'utilisateur revient dans son historique.

Vous pouvez donc utiliser les intentions pour transmettre des messages, mais uniquement dans les situations suivantes :

- Votre récepteur ne se soucie pas de manquer des messages lorsqu'il est inactif.
- Vous fournissez un moyen pour que le récepteur récupère les messages qu'il a manqués pendant qu'il était inactif.

Aux Chapitres 29 et 30, nous verrons un exemple de la première situation, où le récepteur (le client du service) utilise des messages reposant sur des intentions lorsqu'ils sont disponibles, mais n'en a pas besoin quand le client est inactif.

18

Lancement d'activités et de sous-activités

La théorie sous-jacente de l'architecture de l'interface utilisateur d'Android est que les développeurs devraient décomposer leurs applications en activités distinctes, chacune étant implémentée par une `Activity` accessible *via* des intentions, avec une activité "principale" lancée à partir du menu d'Android. Une application de calendrier, par exemple, pourrait avoir des activités permettant de consulter le calendrier, de visualiser un simple événement, d'en modifier un (et d'en ajouter un), etc.

Ceci implique, bien sûr, que l'une de vos activités ait un moyen d'en lancer une autre. Si, par exemple, l'utilisateur clique sur un événement à partir de l'activité qui affiche tout le calendrier, vous voudrez montrer l'activité permettant d'afficher cet événement. Ceci signifie que vous devez pouvoir lancer cette activité en lui faisant afficher un événement spécifique (celui sur lequel l'utilisateur a cliqué).

Info

Ce chapitre suppose que vous connaissez l'activité à lancer, probablement parce qu'elle fait partie de votre application. Vous pouvez également avoir une `Uri` *vers… quelque chose et vous voulez que vos utilisateurs puissent en faire… quelque chose, bien que vous ne sachiez pas encore comment. Cette dernière situation nécessite un traitement plus complexe, que nous décrivons dans l'ouvrage* The Busy Coders Guide to Advanced Android Development *(Commonsware, 2009).*

Activités paires et sous-activités

Lorsque vous décidez de lancer une activité, une question essentielle à laquelle vous devez répondre est : "Est-ce que mon activité a besoin de savoir quand se termine l'activité qu'elle a lancée ?"

Supposons par exemple que vous vouliez créer une activité pour collecter des informations d'authentification pour un service web auquel vous vous connectez – vous devrez peut-être vous authentifier avec OpenID[1] pour utiliser un service OAuth[2]. En ce cas, votre activité principale devra savoir quand se termine l'authentification pour pouvoir commencer à utiliser le service web.

Imaginons maintenant une application de courrier électronique Android. Lorsque l'utilisateur décide de visualiser un fichier attaché, ni vous ni l'utilisateur ne s'attend à ce que l'activité principale sache quand cette visualisation se terminera.

Dans le premier scénario, l'activité lancée est clairement subordonnée à l'activité qui l'a lancée. La première sera donc sûrement lancée comme une sous-activité, ce qui signifie que la seconde sera prévenue de la fin de son activité fille.

Dans le second scénario, l'activité lancée est plutôt un "pair" de l'activité qui l'a lancée. Elle sera donc plutôt lancée comme une activité classique. Votre activité ne sera pas informée de la fin de sa "fille" mais, encore une fois, elle n'a pas vraiment besoin de le savoir.

Démarrage

Pour démarrer une activité, il faut une intention et choisir comment la lancer.

1. **http://openid.net/**.
2. **http://oauth.net/**.

Création d'une intention

Comme on l'a expliqué au chapitre précédent, les intentions encapsulent une requête pour une activité ou une demande adressée à un autre récepteur d'intention, afin qu'il réalise une certaine tâche.

Si l'activité que vous comptez lancer vous appartient, il peut être plus simple de créer une intention explicite, nommant le composant à lancer. À partir de votre activité, vous pourriez par exemple créer une intention de la façon suivante :

```
new Intent(this, HelpActivity.class);
```

Cette instruction indique que vous voulez lancer `HelpActivity`.

Vous pourriez également créer une intention pour une `Uri` donnée, demandant une action particulière :

```
Uri uri=Uri.parse("geo:" + lat.toString() + "," + lon.toString());
Intent i=new Intent(Intent.ACTION_VIEW, uri);
```

Ici, à partir de la latitude et de la longitude d'une position (respectivement `lat` et `lon`), nous construisons une `Uri` de schéma `geo` et nous créons une intention demandant de l'afficher (`ACTION_VIEW`).

Faire appel

Lorsque l'on dispose de l'intention, il faut la passer à Android et récupérer l'activité fille à lancer. Deux choix sont alors possibles :

- Le plus simple consiste à appeler `startActivity()` en lui passant l'intention – Android recherchera l'activité qui correspond le mieux et lui passera l'intention pour qu'elle la traite. Votre activité ne sera pas prévenue de la fin de l'activité fille.
- Vous pouvez aussi appeler `startActivityForResult()` en lui passant l'intention et un identifiant (unique pour l'activité appelante). Android recherchera l'activité qui correspond le mieux et lui passera l'intention. Votre activité sera prévenue par la méthode de rappel `onActivityResult()` de la fin de l'activité fille.

Comme on l'a indiqué, vous pouvez implémenter la méthode de rappel `onActivityResult()` lorsque vous utilisez `startActivityForResult()`, afin d'être prévenu de la fin de l'activité fille. Cette méthode reçoit l'identifiant unique fourni à `startActivityForResult()` pour que vous puissiez savoir quelle est l'activité qui s'est terminée. Vous récupérez également :

- Le code résultat de l'activité fille qui a appelé `setResult()`. Généralement, ce code vaut `RESULT_OK` ou `RESULT_CANCELLED`, bien que vous puissiez créer vos propres codes (choisissez un entier à partir de la valeur `RESULT_FIRST_USER`).

- Un objet `String` facultatif contenant des données du résultat, comme une URL vers une ressource interne ou externe – une intention `ACTION_PICK` se sert généralement de cette chaîne pour renvoyer le contenu sélectionné.
- Un objet `Bundle` facultatif contenant des informations supplémentaires autres que le code résultat et la chaîne de données.

Pour mieux comprendre le lancement d'une activité paire, examinons le projet `Activities/Launch`. Le fichier de description XML est assez simple puisqu'il contient deux champs pour la latitude et la longitude, ainsi qu'un bouton :

```
<?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"
  >
  <TableLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:stretchColumns="1,2"
  >
    <TableRow>
      <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingLeft="2dip"
        android:paddingRight="4dip"
        android:text="Emplacement : "
      />
      <EditText android:id="@+id/lat"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:cursorVisible="true"
        android:editable="true"
        android:singleLine="true"
        android:layout_weight="1"
      />
      <EditText android:id="@+id/lon"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:cursorVisible="true"
        android:editable="true"
        android:singleLine="true"
        android:layout_weight="1"
      />
    </TableRow>
```

```
  </TableLayout>
  <Button android:id="@+id/map"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Afficher !"
  />
</LinearLayout>
```

L'`OnClickListener` du bouton prend la latitude et la longitude pour les intégrer dans une `Uri` de schéma `geo`, puis lance l'activité.

```
package com.commonsware.android.activities;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;

public class LaunchDemo extends Activity {
  private EditText lat;
  private EditText lon;

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

    Button btn=(Button)findViewById(R.id.map);
    lat=(EditText)findViewById(R.id.lat);
    lon=(EditText)findViewById(R.id.lon);

    btn.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        String _lat=lat.getText().toString();
        String _lon=lon.getText().toString();
        Uri uri=Uri.parse("geo:" + _lat + "," +_lon);

        startActivity(new Intent(Intent.ACTION_VIEW, uri));
      }
    });
  }
}
```

Comme on peut le constater avec la Figure 18.1, cette activité ne montre pas grand-chose lorsqu'elle est lancée.

Figure 18.1
L'application `LaunchDemo`*, dans laquelle on a saisi un emplacement.*

L'affichage devient plus intéressant si l'on entre un emplacement (48.85825 de latitude et 2.2945 de longitude, par exemple) et que l'on clique sur le bouton (voir Figure 18.2). Notez qu'il s'agit de l'activité de cartographie intégrée à Android : nous n'avons pas créé d'activité pour afficher cette carte.

Au Chapitre 33, nous verrons comment créer des cartes dans nos activités, au cas où l'on aurait besoin de plus de contrôle sur leur affichage.

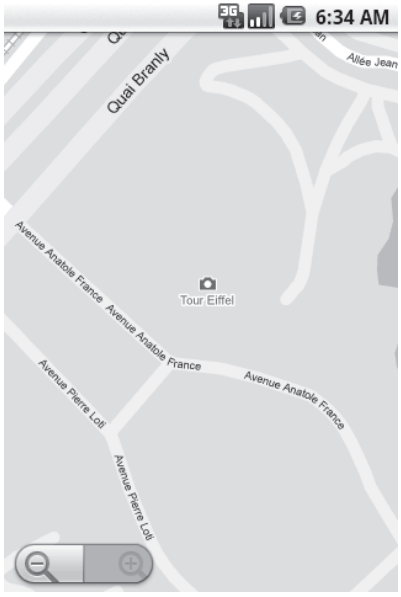


Figure 18.2
La carte lancée par `LaunchDemo`*, montrant l'emplacement de la tour Eiffel, à Paris.*

Cet exemple risque de ne pas fonctionner dans l'émulateur avec un AVD Android 2.0 car cet AVD ne dispose pas de l'application Maps.

Navigation avec onglets

La navigation par onglets est l'une des principales fonctionnalités des navigateurs web actuels : grâce à elle, une même fenêtre peut afficher plusieurs pages réparties dans une série d'onglets. Sur un terminal mobile, elle a moins d'intérêt car on gaspillerait la précieuse surface de l'écran pour afficher les onglets eux-mêmes. Toutefois, pour les besoins de la démonstration, nous montrerons comment créer ce genre de navigateur, en utilisant `TabActivity` et les intentions.

Au Chapitre 9, nous avons vu qu'un onglet pouvait contenir une vue ou une activité. Dans ce dernier cas, vous devez fournir une intention qui lancera l'activité souhaitée ; le framework de gestion des onglets placera alors l'interface utilisateur de cette activité dans l'onglet.

Votre premier instinct pourrait être d'utiliser une `Uri http:` comme nous l'avions fait avec une `Uri geo:` dans l'exemple précédent :

```
Intent i=new Intent(Intent.ACTION_VIEW);
i.setData(Uri.parse("http://www.pearson.fr"));
```

Vous pourriez ainsi utiliser le navigateur intégré et disposer de toutes ses fonctionnalités.

Malheureusement, cela ne marche pas car, pour des raisons de sécurité, vous ne pouvez pas héberger les activités d'autres applications dans vos onglets – uniquement vos propres activités.

Nous allons donc dépoussiérer nos démonstrations de `WebView` du Chapitre 13 pour créer le projet `Activities/IntentTab`.

Voici le code source de l'activité principale, celle qui héberge le `TabView` :

```
public class IntentTabDemo extends TabActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    TabHost host=getTabHost();

    host.addTab(host.newTabSpec("un")
            .setIndicator("CW")
```

```
            .setContent(new Intent(this, CWBrowser.class)));
    host.addTab(host.newTabSpec("deux")
            .setIndicator("Android")
            .setContent(new Intent(this, AndroidBrowser.class)));
  }
}
```

Comme vous pouvez le constater, notre classe hérite de `TabActivity` : nous n'avons donc pas besoin de créer un fichier de description XML car `TabActivity` s'en occupe pour nous. Nous nous contentons d'accéder au `TabHost` et de lui ajouter deux onglets, chacun précisant une intention qui fait directement référence à une autre classe. Ici, nos deux onglets hébergeront respectivement un `CWBrowser` et un `AndroidBrowser`.

Ces activités sont de simples variantes de nos précédents exemples de navigateurs :

```
public class CWBrowser extends Activity {
  WebView browser;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    browser=new WebView(this);
    setContentView(browser);
    browser.loadUrl("http://www.pearson.fr");
  }
}
public class AndroidBrowser extends Activity {
  WebView browser;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    browser=new WebView(this);
    setContentView(browser);
    browser.loadUrl("http://www.android.com");
  }
}
```

Tous les deux chargent simplement une URL différente dans le navigateur : la page d'accueil de Pearson France dans l'un (voir Figure 18.3), celle d'Android dans l'autre (voir Figure 18.4). Le résultat montre l'aspect qu'aurait un navigateur à onglets avec Android.

Figure 18.3
L'application `IntentTabDemo` *montrant le premier onglet.*

Figure 18.4
L'application `IntentTabDemo` *montrant le second onglet.*

L'utilisation de sous-classes différentes pour chaque page cible est plutôt onéreuse. À la place, nous aurions pu empaqueter l'URL pour qu'elle s'ouvre comme un "extra" dans une intention et utiliser cette intention pour créer une activité `BrowserTab` *généraliste qui extrairait l'URL de cet "extra" afin de l'utiliser.*

19

Gestion de la rotation

Certains terminaux Android comme le G1 disposent d'un clavier "à tiroir" qui, lorsqu'il est sorti, provoque le passage de l'écran du mode portrait au mode paysage. D'autres, comme l'iPhone, utilisent des accéléromètres pour déterminer l'orientation de l'écran.

Android fournit plusieurs moyens de gérer la rotation de l'écran afin que vos applications puissent elles-mêmes traiter correctement les deux orientations. Ces outils vous aident simplement à détecter et à gérer le processus de rotation – c'est à vous de vérifier que vos interfaces utilisateurs apparaîtront correctement dans les deux orientations.

Philosophie de la destruction

Par défaut, lorsqu'une modification dans la configuration du téléphone risque d'affecter la sélection des ressources, Android supprimera et recréera toutes les activités en cours d'exécution ou en pause la prochaine fois qu'elles seront affichées. Bien que ce phénomène puisse avoir lieu pour un grand nombre de modifications de configuration (changement de la langue, par exemple), il interviendra surtout dans le cas des rotations, car un pivotement de l'écran force le chargement d'un nouvel ensemble de ressources (les layouts, notamment).

Il faut bien comprendre qu'il s'agit du comportement par défaut : il peut être le plus adapté à l'une ou l'autre de vos activités, mais vous pouvez le contrôler et adapter la réponse de vos activités aux changements d'orientation ou de configuration.

Tout est pareil, juste différent

Comme, par défaut, Android détruit et récrée votre activité lorsque l'orientation change, il suffit d'utiliser la méthode `onSaveInstanceState()`, qui est appelée à chaque fois que l'activité est supprimée quelle qu'en soit la raison (un manque de mémoire, par exemple). Implémentez cette méthode dans votre activité afin de stocker dans le `Bundle` fourni suffisamment d'informations pour pouvoir revenir à l'état courant. Puis, dans `onCreate()` (ou `onRestoreInstanceState()`, si vous préférez), restaurez les données du `Bundle` et utilisez-les pour remettre l'activité dans son état antérieur.

Le projet `Rotation/RotationOne`, par exemple, utilise deux fichiers layout `main.xml` pour les modes portrait et paysage. Ces fichiers se trouvent, respectivement, dans les répertoires `res/layout/` et `res/layout-land/`.

Voici la disposition du mode portrait :

```
<?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/pick"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1"
    android:text="Choisir"
    android:enabled="true"
  />
  <Button android:id="@+id/view"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1"
    android:text="Voir"
    android:enabled="false"
  />
</LinearLayout>
```

Voici celle du mode paysage :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <Button android:id="@+id/pick"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1"
    android:text="Choisir"
    android:enabled="true"
  />
  <Button android:id="@+id/view"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1"
    android:text="Voir"
    android:enabled="false"
  />
</LinearLayout>
```

L'interface utilisateur est essentiellement composée de deux boutons occupant, chacun, la moitié de l'écran. En mode portrait, les boutons sont placés l'un au-dessus de l'autre ; en mode paysage, ils sont côte à côte.

L'application semble fonctionner correctement – une rotation (Ctrl+F12 dans l'émulateur) modifie la disposition de l'interface. Bien que les boutons n'aient pas d'état, vous constateriez, si vous utilisiez d'autres widgets (comme `EditText`), qu'Android prend automatiquement en charge une partie de leur état (le texte saisi dans l'`EditText`, par exemple).

En revanche, Android ne peut pas vous aider pour tout ce qui se trouve à l'extérieur des widgets.

Cette application permet de choisir un contact et de consulter sa fiche au moyen de deux boutons différents. Le bouton Voir n'est actif que lorsqu'un contact a été sélectionné.

Voyons comment tout ceci est géré par `onSaveInstanceState()` :

```
public class RotationOneDemo extends Activity {
  static final int PICK_REQUEST=1337;
  Button viewButton=null;
  Uri contact=null;
```

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

  Button btn=(Button)findViewById(R.id.pick);

  btn.setOnClickListener(new View.OnClickListener() {
    public void onClick(View view) {
      Intent i=new Intent(Intent.ACTION_PICK,
                  Uri.parse("content://contacts/people"));
      startActivityForResult(i, PICK_REQUEST);
    }
  });

  viewButton=(Button)findViewById(R.id.view);

  viewButton.setOnClickListener(new View.OnClickListener() {
    public void onClick(View view) {
      startActivity(new Intent(Intent.ACTION_VIEW, contact));
    }
  });

  restoreMe(savedInstanceState);

  viewButton.setEnabled(contact!=null);
}
@Override
protected void onActivityResult(int requestCode, int resultCode,
                                Intent data) {
  if (requestCode==PICK_REQUEST) {
    if (resultCode==RESULT_OK) {
      contact=data.getData();
      viewButton.setEnabled(true);
    }
  }
}
@Override
protected void onSaveInstanceState(Bundle outState) {
  super.onSaveInstanceState(outState);
```

```
    if (contact!=null) {
      outState.putString("contact", contact.toString());
    }
  }

  private void restoreMe(Bundle state) {
    contact=null;

    if (state!=null) {
      String contactUri=state.getString("contact");

      if (contactUri!=null) {
        contact=Uri.parse(contactUri);
      }
    }
  }
}
```

Dans l'ensemble, ceci ressemble à une activité normale... parce que c'en est une. Initialement, le "modèle" – un contact désigné par une `Uri` – vaut `null`. Il est initialisé à la suite du lancement de la sous-activité `ACTION_PICK`. Sa représentation sous forme de chaîne est sauvegardée dans `onSaveInstanceState()` et restaurée dans `restoreMe()` (qui est appelée à partir d'`onCreate()`). Si le contact n'est pas `null`, le bouton Voir est activé et permet de visualiser la fiche du contact sélectionné.

Le résultat est présenté aux Figures 19.1 et 19.2.

Figure 19.1
L'application `RotationOne` *en mode portrait.*

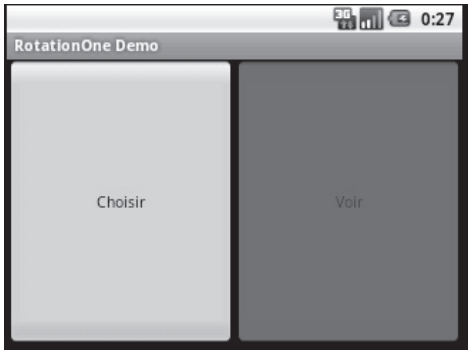


Figure 19.2
L'application `RotationOne` *en mode paysage.*

L'avantage de cette implémentation est qu'elle gère un certain nombre d'événements système en plus de la rotation – la fermeture de l'application à cause d'un manque de mémoire, par exemple.

Pour le plaisir, mettez en commentaire l'appel de `restoreMe()` dans `onCreate()` et essayez de lancer l'application : vous constaterez qu'elle "oublie" un contact sélectionné dans l'une des orientations lorsque vous faites tourner l'émulateur ou le terminal.

Les exemples de ce chapitre ne fonctionnent qu'avec Android 2.0 ou une version supérieure car ils utilisent les nouveaux mécanismes de sélection du fournisseur de contenu `Contacts` *(voir Chapitre 26).*

Il n'y a pas de petites économies !

Le problème avec `onSaveInstanceState()` est que l'on est limité à un `Bundle` car cette méthode de rappel est également appelée lorsque tout le processus est arrêté (comme quand il n'y a plus assez de mémoire) : les données à sauvegarder doivent donc pouvoir être sérialisées et ne pas dépendre du processus en cours.

Pour certaines activités, cette restriction ne pose pas de problème mais, pour d'autres, elle peut être assez ennuyeuse. Dans une application de messagerie instantanée, par exemple, vous devrez couper la connexion au serveur et la rétablir car il n'y a pas moyen de stocker une socket dans un `Bundle` ; ceci ne pose pas seulement un problème de performances mais peut également affecter la discussion elle-même : votre déconnexion et votre reconnexion apparaîtront dans les journaux du chat.

Un moyen de contourner ce problème consiste à utiliser `onRetainNonConfigurationInstance()` au lieu d'`onSaveInstanceState()` pour les "petites" modifications comme les rotations. La méthode de rappel `onRetainNonConfigurationInstance()` de votre activité

peut en effet renvoyer un `Object`, que vous pourrez récupérer plus tard avec `getLastNonConfigurationInstance()`. Cet `Object` peut représenter à peu près tout ce que vous voulez – généralement, ce sera un objet "contexte" contenant l'état de l'activité, comme les threads en cours d'exécution, les sockets ouvertes, etc. La méthode `onCreate()` de votre activité peut appeler `getLastNonConfigurationInstance()` – si elle renvoie une valeur non `null`, vous disposez de vos sockets, de vos threads, etc. La plus grande limitation est que vous ne pouvez pas mettre dans ce contexte sauvegardé tout ce qui pourrait faire référence à une ressource qui sera supprimée, comme un `Drawable` chargé à partir d'une ressource.

Le projet `Rotation/RotationTwo` utilise cette approche pour gérer les rotations. Les fichiers de description, et donc l'apparence visuelle, sont identiques à ceux de `Rotation/RotationOne`. Les différences apparaissent uniquement dans le code Java :

```
public class RotationTwoDemo extends Activity {
  static final int PICK_REQUEST=1337;
  Button viewButton=null;
  Uri contact=null;

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

    Button btn=(Button)findViewById(R.id.pick);

    btn.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        Intent i=new Intent(Intent.ACTION_PICK,
                    Uri.parse("content://contacts/people"));
        startActivityForResult(i, PICK_REQUEST);
      }
    });

    viewButton=(Button)findViewById(R.id.view);

    viewButton.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        startActivity(new Intent(Intent.ACTION_VIEW, contact));
      }
    });
```

```
    restoreMe();

    viewButton.setEnabled(contact!=null);
  }
  @Override
  protected void onActivityResult(int requestCode, int resultCode,
                                  Intent data) {
    if (requestCode==PICK_REQUEST) {
      if (resultCode==RESULT_OK) {
        contact=data.getData();
        viewButton.setEnabled(true);
      }
    }
  }
  @Override
  public Object onRetainNonConfigurationInstance() {
    return(contact);
  }

  private void restoreMe() {
    contact=null;

    if (getLastNonConfigurationInstance()!=null) {
      contact=(Uri)getLastNonConfigurationInstance();
    }
  }
}
```

Ici, nous redéfinissons `onRetainNonConfigurationInstance()` pour qu'elle renvoie l'`Uri` de notre contact au lieu de sa représentation textuelle. De son côté, `restoreMe()` appelle `getLastNonConfigurationInstance()` : si cette méthode renvoie une valeur non `null`, il s'agit de notre contact et l'on active le bouton Voir.

L'avantage de cette approche est que l'on transmet l'`Uri` au lieu d'une représentation textuelle. Ici, cela ne représente pas une grosse économie, mais notre état pourrait être bien plus compliqué et contenir des threads, des sockets ou d'autres choses que nous ne pourrions pas empaqueter dans un `Bundle`.

Ceci dit, même cette approche peut être trop intrusive pour votre application. Supposons, par exemple, que vous développiez un jeu en temps réel, comme un FPS (*First Person Shooter*). Les "ralentissements" (*lags*) que vos utilisateurs constateront lorsque l'activité est supprimée et recréée peuvent leur suffire à se faire tuer, ce qu'ils n'apprécieront sûrement pas. Bien que ce soit moins un problème avec le G1 puisqu'une rotation implique d'ouvrir le clavier – chose que l'utilisateur ne fera pas au milieu d'un jeu –, d'autres terminaux

effectuent une rotation simplement en fonction des informations de leurs accéléromètres. Dans ce cas, vous pouvez indiquer à Android que vous vous occuperez vous-même des rotations et que vous ne voulez pas que le framework vous aide.

Rotation maison

Pour gérer vous-même les rotations, vous devez procéder comme suit :

1. Utilisez l'attribut `android:configChanges` de l'élément `activity` dans votre fichier `AndroidManifest.xml` pour énumérer les modifications de configuration que vous voulez gérer vous-même.
2. Dans votre classe `Activity`, implémentez la méthode `onConfigurationChanged()`, qui sera appelée lorsque l'une des modifications énumérées dans `android:configChanges` aura lieu.

Vous pouvez désormais outrepasser tout le processus de suppression d'activité pour n'importe quelle modification de configuration et simplement laisser une méthode de rappel vous prévenir de cette modification.

Le projet `Rotation/RotationThree` utilise cette technique. Là encore, les fichiers de description sont les mêmes que précédemment et l'application aura donc le même aspect que `RotationOne` et `RotationTwo`. Le code Java est en revanche assez différent car nous nous préoccupons non plus de sauvegarder l'état mais plutôt de mettre à jour l'interface utilisateur pour gérer les changements d'orientation.

Nous devons d'abord ajouter une petite modification à notre manifeste :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.commonsware.android.rotation.three"
      android:versionCode="1"
      android:versionName="1.0.0">
  <uses-sdk
      android:minSdkVersion="5"
      android:targetSdkVersion="6"
  />
    <application android:label="@string/app_name">
        <activity android:name=".RotationThreeDemo"
                  android:label="@string/app_name"
                  android:configChanges="keyboardHidden|orientation">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
```

```
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Ici, nous indiquons que nous traiterons nous-mêmes les modifications de configuration `keyboardHidden` et `orientation`, ce qui nous protège de toutes les modifications de "rotation" – une ouverture d'un clavier ou une rotation physique. Notez que cette configuration s'applique à une activité, pas à l'application – si vous avez plusieurs activités, vous devrez décider pour chacune d'elles de la tactique à employer.

Voici le code Java de ce projet :

```
public class RotationThreeDemo extends Activity {
  static final int PICK_REQUEST=1337;
  Button viewButton=null;
  Uri contact=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setupViews();
  }
  @Override
  protected void onActivityResult(int requestCode, int resultCode,
                                  Intent data) {
    if (requestCode==PICK_REQUEST) {
      if (resultCode==RESULT_OK) {
        contact=data.getData();
        viewButton.setEnabled(true);
      }
    }
  }
  public void onConfigurationChanged(Configuration newConfig) {
    super.onConfigurationChanged(newConfig);

    setupViews();
  }

  private void setupViews() {
    setContentView(R.layout.main);
```

```
        Button btn=(Button)findViewById(R.id.pick);

        btn.setOnClickListener(new View.OnClickListener() {
          public void onClick(View view) {
            Intent i=new Intent(Intent.ACTION_PICK,
                        Uri.parse("content://contacts/people"));
            startActivityForResult(i, PICK_REQUEST);
          }
        });

        viewButton=(Button)findViewById(R.id.view);

        viewButton.setOnClickListener(new View.OnClickListener() {
          public void onClick(View view) {
            startActivity(new Intent(Intent.ACTION_VIEW, contact));
          }
        });

        viewButton.setEnabled(contact!=null);
      }
    }
```

L'implémentation d'`onCreate()` délègue l'essentiel de son traitement à la méthode `setupViews()`, qui charge le layout et configure les boutons. Cette partie a été placée dans sa propre méthode car elle est également appelée à partir d'`onConfigurationChanged()`.

Forcer le destin

Dans les trois sections précédentes, nous avons vu comment traiter les événements de rotation. Il existe, bien sûr, une alternative radicale : demander à Android de ne jamais faire pivoter votre activité car, si l'activité ne pivote pas, il n'est plus nécessaire de se soucier d'écrire le code pour gérer les rotations.

Pour empêcher Android de faire pivoter votre activité, il suffit d'ajouter `android:screenOrientation = "portrait"` (ou `"landscape"`) au fichier `AndroidManifest.xml` (voir le projet `Rotation/RotationFour`) :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.commonsware.android.rotation.four"
      android:versionCode="1"
```

```
        android:versionName="1.0.0">
    <uses-sdk
        android:minSdkVersion="5"
        android:targetSdkVersion="6"
    />
      <application android:label="@string/app_name">
          <activity android:name=".RotationFourDemo"
                    android:screenOrientation="portrait"
                    android:label="@string/app_name">
              <intent-filter>
                  <action android:name="android.intent.action.MAIN" />
                  <category android:name="android.intent.category.LAUNCHER" />
              </intent-filter>
          </activity>
      </application>
  </manifest>
```

N'oubliez pas que, comme précédemment, cette configuration est propre à une activité.

Avec elle, l'activité est verrouillée dans l'orientation que vous avez précisée, quoi que l'on fasse ensuite. Les copies d'écran des Figures 19.3 et 19.4 montrent la même activité que celle des trois sections précédentes mais utilisent le manifeste ci-dessus, avec l'émulateur en mode portrait et en mode paysage. Vous remarquerez que l'interface utilisateur n'a pas varié et qu'elle reste en mode portrait dans les deux cas.

Figure 19.3
L'application `RotationFour` *en mode portrait.*

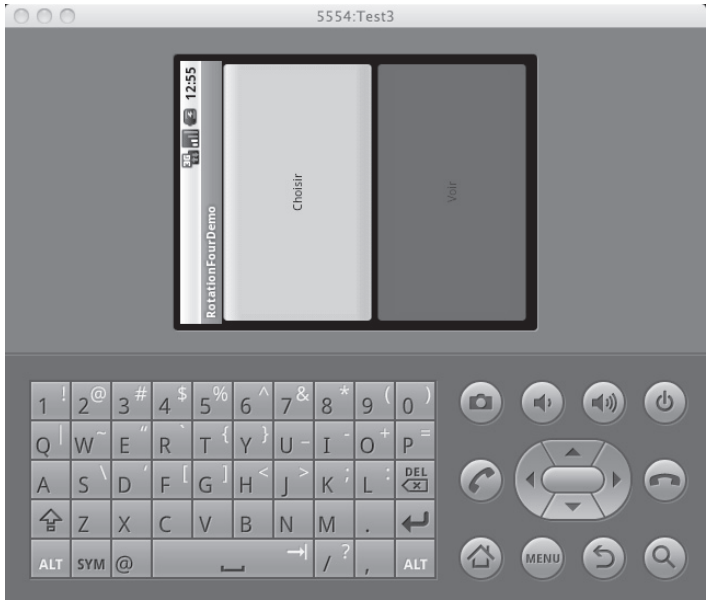


Figure 19.4
L'application `RotationFour` *en mode paysage.*

Notez qu'Android supprimera et recréera quand même votre activité, même si l'orientation est fixée à une valeur spécifique, comme ici. Pour éviter cela, il faut également initialiser `android:configChanges` dans le manifeste, comme expliqué plus haut.

Tout comprendre

Tous ces scénarios supposent que vous faites pivoter l'écran en ouvrant le clavier du terminal (ou en utilisant la combinaison de touches Ctrl+F12 avec l'émulateur), ce qui est la norme pour les applications Android.

Cependant, nous n'avons pas encore présenté le scénario utilisé par l'iPhone.

Vous avez sans doute déjà vu une ou plusieurs publicités pour l'iPhone montrant comment l'écran change d'orientation lorsque l'on fait pivoter le téléphone. Certains téléphones Android, comme le HTC Magic, se comportent de la même façon ; pour d'autres, l'orientation de l'écran ne dépend que de l'ouverture ou de la fermeture du clavier.

Cependant, même pour ces derniers, il est très facile de modifier ce comportement : pour que l'affichage pivote en fonction de la position du téléphone, il suffit d'ajouter `android:screenOrientation = "sensor"` au fichier `AndroidManifest.xml` (voir le projet `Rotation/RotationFive`) :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
```

```
        package="com.commonsware.android.rotation.five"
        android:versionCode="1"
        android:versionName="1.0.0">
    <uses-sdk
        android:minSdkVersion="5"
        android:targetSdkVersion="6"
    />
      <application android:label="@string/app_name">
          <activity android:name=".RotationFiveDemo"
                    android:screenOrientation="sensor"
                    android:label="@string/app_name">
              <intent-filter>
                  <action android:name="android.intent.action.MAIN" />
                  <category android:name="android.intent.category.LAUNCHER" />
              </intent-filter>
          </activity>
      </application>
  </manifest>
```

La valeur `"sensor"` indique à Android que vous souhaitez que ce soient les accéléromètres qui contrôlent l'orientation de l'écran, afin qu'il pivote en même temps que le téléphone.

Sur le G1, en tout cas, ceci semble ne fonctionner que lorsque l'on passe de la position portrait classique à la position paysage – en faisant pivoter le téléphone de 90 degrés dans le sens inverse des aiguilles d'une montre. Une rotation dans l'autre sens ne modifie pas l'écran.

Notez également que cette configuration désactive la rotation de l'écran par l'ouverture du clavier. Dans une activité "normale", avec le terminal en position portrait, l'écran pivotera si l'on ouvre le clavier ; avec une activité utilisant la configuration `android:screenOrientation = "sensor"`, l'écran ne pivotera pas.

20

Utilisation des ressources

Les ressources sont des informations statiques, stockées en dehors du code Java. Dans les exemples de ce livre, vous avez déjà souvent rencontré un type de ressource – les fichiers de description (*layouts*) –, mais il en existe de nombreux autres dont vos applications peuvent tirer profit : les images et les chaînes de caractères, par exemple.

Les différents types de ressources

Les ressources d'un projet Android sont stockées dans des fichiers situés sous le répertoire `res/` de l'arborescence. À l'exception des ressources brutes (`res/raw/`), tous les types de ressources sont analysés automatiquement, soit par le système de paquetages d'Android, soit par le système du terminal ou de l'émulateur. Si vous décrivez, par exemple, l'interface utilisateur d'une activité *via* une ressource de type layout (dans `res/layout`), vous n'avez pas besoin d'analyser vous-même le contenu du fichier XML – Android s'en chargera pour vous.

Outre les layouts (que nous avons rencontrés pour la première fois au Chapitre 4) et les animations (introduites au Chapitre 9), il existe plusieurs autres types de ressources :

- Les images (`res/drawable`) permettent de placer des icônes statiques ou d'autres images dans une interface utilisateur.
- Les ressources brutes (`res/raw`) permettent d'utiliser des fichiers quelconques ayant une signification pour votre application, mais pas nécessairement pour les frameworks d'Android.
- Les chaînes, les couleurs, les tableaux et les dimensions (`res/values/`) permettent d'associer des noms symboliques à ces types de constantes et de les séparer du reste du code (pour l'internationalisation et la localisation, notamment).
- Les fichiers XML statiques (`res/xml/`) permettent de stocker vos propres données et structures.

Théorie des chaînes

Placer les labels et les autres textes à l'extérieur du code source de l'application est, généralement, une très bonne idée. Ceci facilite, notamment, l'internationalisation (*I18N*) et la localisation (*L10N*), qui sont présentées un peu plus loin dans ce chapitre. Même si vous ne comptez pas traduire vos textes dans d'autres langues, cette séparation facilite les corrections car toutes les chaînes sont au même endroit au lieu d'être disséminées dans le code source.

Android permet d'utiliser des chaînes externes classiques, mais également des "formats de chaînes", contenant des emplacements qui seront remplacés par les véritables informations au cours de l'exécution. En outre, le formatage de texte simple, appelé "texte stylé", permet de mélanger des mots en gras ou en italique avec du texte normal.

Chaînes normales

En règle générale, vous n'avez besoin pour les chaînes normales que d'un fichier XML situé dans le répertoire `res/values` (le plus souvent `res/values/strings.xml`). La racine de ce document est l'élément `ressources`, qui a autant de fils `string` qu'il y a de chaînes à encoder comme ressource. L'élément `string` a un attribut `name` contenant le nom unique de la chaîne. Le contenu de cet élément est le texte de la chaîne :

```
<resources>
  <string name="whisky">Portez ce vieux whisky...</string>
  <string name="zephir">Le vif zéphir jubile...</string>
</resources>
```

Le seul point épineux concerne la présence de guillemets (`"`) ou d'apostrophes (`'`) dans le texte de la chaîne car vous devrez alors les protéger en les préfixant d'un antislash (`il fait beau aujourd\'hui`, par exemple). Dans le cas de l'apostrophe, vous pouvez également placer tout le texte entre guillemets (`"il fait beau aujourd'hui"`).

Vous pouvez ensuite faire référence à cette chaîne depuis un fichier de description (sous la forme `@string/whisky`, ou `@string/zephir`, par exemple) ou y accéder depuis votre code Java à l'aide de `getString()`, en lui passant l'identifiant de ressource de la chaîne, c'est-à-dire son nom unique préfixé par `R.string` (comme `getString(R.string.whisky)`).

Formats de chaînes

Comme les autres implémentations du langage Java, la machine virtuelle Dalvik d'Android reconnaît les formats de chaînes. Ces formats sont des chaînes contenant des marqueurs d'emplacements et seront remplacés lors de l'exécution par des données variables (`Mon nom est %1$s`, par exemple). Vous pouvez utiliser pour ces formats des chaînes normales stockées sous forme de ressources :

```
String strFormat=getString(R.string.mon_nom);
String strResult=String.format(strFormat, "Tim");
((TextView)findViewById(R.id.un_label)).setText(strResult);
```

Texte stylé

Pour enrichir du texte, vous pourriez utiliser des ressources brutes contenant du HTML, puis les placer dans un widget `WebKit`. Cependant, pour un formatage léger utilisant `<b>`, `<i>` et `<u>`, une ressource chaîne fera l'affaire. Par contre, vous devez protéger les marqueurs HTML et non les traiter littéralement :

```
<resources>
  <string name="b">Ce texte est en &lt;b&gt;gras&lt;/b&gt;.</string>
  <string name="i">
     Alors que celui-ci est en &lt;i&gt;italiques&lt;/i&gt; !
  </string>
</resources>
```

Vous pouvez ensuite y accéder comme n'importe quelle autre chaîne normale, sauf que le résultat de l'appel à `getString()` sera ici un objet implémentant l'interface `android.text.Spanned` :

```
((TextView)findViewById(R.id.autre_label))
            .setText(getString(R.string.i));
```

Formats stylés

Le texte stylé devient compliqué à gérer lorsqu'il s'agit de l'utiliser avec les formats de chaînes car `String.format()` s'applique à des objets `String`, pas à des objets `Spanned` disposant d'instructions de formatage. Si vous avez vraiment besoin de formats de chaînes stylés, vous pouvez suivre ces étapes :

1. Dans la ressource chaîne, remplacez les chevrons par des entités HTML (`Je suis <b>%1$s</b>`, par exemple).
2. Récupérez la ressource comme d'habitude, bien qu'elle ne soit pas encore stylée à ce stade (avec `getString(R.string.format_style)`).
3. Produisez le résultat du formatage, en vous assurant de protéger les valeurs de chaînes que vous substituez, au cas où elles contiendraient des chevrons ou des esperluettes :

```
String.format(getString(R.string.format_style),
              TextUtils.htmlEncode(nom));
```

4. Convertissez le HTML encodé vers un objet `Spanned`, grâce à `Html.fromHtml()`.

```
uneTextView.setText(Html.fromHtml(resultatDeStringFormat));
```

Pour voir tout ceci en action, examinons le fichier de description du projet `Resources/Strings` :

```
<?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"
  >
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    >
    <Button android:id="@+id/format"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/btn_name"
      />
    <EditText android:id="@+id/name"
```

```
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        />
    </LinearLayout>
    <TextView android:id="@+id/result"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      />
</LinearLayout>
```

Comme vous pouvez le constater, l'interface utilisateur n'est composée que d'un bouton, d'un champ et d'un label. Le but est que l'utilisateur entre son nom dans le champ puis clique sur le bouton pour que le label soit remplacé par un message formaté contenant ce nom.

L'élément `Button` de ce fichier faisant référence à une ressource chaîne (`@string/btn_name`), nous avons besoin d'un fichier de ressource chaîne (`res/values/strings.xml`) :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">StringsDemo</string>
  <string name="btn_name">Nom : </string>
  <string name="funky_format">Je suis &lt;b&gt;%1$s&lt;/b&gt;</string>
</resources>
```

La ressource `app_name` est automatiquement créée par le script `activityCreator`. La chaîne `btn_name` est le titre du bouton, tandis que le format de chaîne stylé se trouve dans `funky_format`.

Enfin, nous avons besoin d'un peu de Java pour relier tous les morceaux :

```
package com.commonsware.android.resources;

import android.app.Activity;
import android.os.Bundle;
import android.text.TextUtils;
import android.text.Html;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

public class StringsDemo extends Activity {
```

```
    EditText name;
    TextView result;

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

      name=(EditText)findViewById(R.id.name);
      result=(TextView)findViewById(R.id.result);

      Button btn=(Button)findViewById(R.id.format);

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

    private void applyFormat() {
      String format=getString(R.string.funky_format);
      String simpleResult=String.format(format,
                        TextUtils.htmlEncode(name.getText().toString()));
      result.setText(Html.fromHtml(simpleResult));
    }
  }
```

La manipulation de la ressource chaîne a lieu dans `applyFormat()`, qui est appelée lorsque l'on clique sur le bouton. On récupère d'abord notre format par un appel à `getString()` – chose que nous aurions pu faire dans `onCreate()` pour plus d'efficacité. Puis nous nous en servons pour formater la valeur du champ, ce qui nous renvoie un `String` puisque la ressource chaîne est encodée en HTML. Vous remarquerez que nous utilisons `TextUtils.htmlEncode()` pour traduire en entités les caractères spéciaux du nom qui a été saisi, au cas où un utilisateur déciderait d'entrer une esperluette, par exemple. Enfin, on convertit ce texte HTML en objet texte stylé à l'aide de `Html.fromHtml()` et nous modifions notre label.

La Figure 20.1 montre que le label de l'activité est vide lors de son lancement.

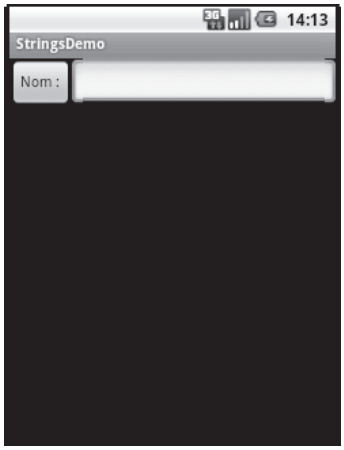


Figure 20.1
L'application `StringsDemo` *après son lancement.*

La Figure 20.2 montre ce que l'on obtient après avoir saisi un nom et cliqué sur le bouton.

Figure 20.2
La même application, après avoir entré un nom célèbre.

Vous voulez gagner une image ?

Android reconnaît les images aux formats PNG, JPEG et GIF, bien que GIF soit officiellement déconseillé ; il est généralement préférable d'utiliser PNG. Les images peuvent être utilisées partout où l'on attend un objet `Drawable`, comme pour l'image et le fond d'une `ImageView`.

L'utilisation des images consiste simplement à placer les fichiers images dans `res/drawable/` puis à y faire référence comme des ressources. Dans les fichiers de descrip-

tion XML, vous pouvez les désigner par `@drawable/`*`nomfic`*, où *nomfic* est le nom de base du fichier (le nom de la ressource correspondant au fichier `res/drawable/truc.png` est donc `@drawable/truc`). Dans le code Java, il suffit de préfixer le nom de base du fichier par `R.drawable` lorsque vous avez besoin de l'identifiant de ressource (`R.drawable.truc`, par exemple).

Modifions l'exemple précédent pour qu'il utilise une icône à la place de la ressource chaîne du bouton. Ce projet, `Resources/Images`, nécessite d'abord de modifier légèrement le fichier de description afin d'utiliser un `ImageButton` et de faire référence à un `Drawable` nommé `@drawable/icon` :

```
<?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"
  >
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    >
    <ImageButton android:id="@+id/format"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:src="@drawable/icon"
      />
    <EditText android:id="@+id/name"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      />
  </LinearLayout>
  <TextView android:id="@+id/result"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    />
</LinearLayout>
```

Puis nous devons placer un fichier image dans `res/drawable` avec `icon` comme nom de base. Ici, nous utiliserons un fichier PNG de 32 × 32 pixels, issu de l'ensemble d'icônes Nuvola[1]. Enfin, nous modifions le code Java pour remplacer le `Button` par un `ImageButton` :

1. **http://en.wikipedia.org/wiki/Nuvola**.

```
package com.commonsware.android.resources;

import android.app.Activity;
import android.os.Bundle;
import android.text.TextUtils;
import android.text.Html;
import android.view.View;
import android.widget.Button;
import android.widget.ImageButton;
import android.widget.EditText;
import android.widget.TextView;

public class ImagesDemo extends Activity {
  EditText name;
  TextView result;

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

    name=(EditText)findViewById(R.id.name);
    result=(TextView)findViewById(R.id.result);

    ImageButton btn=(ImageButton)findViewById(R.id.format);

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

  private void applyFormat() {
    String format=getString(R.string.funky_format);
    String simpleResult=String.format(format,
                    TextUtils.htmlEncode(name.getText().toString()));
    result.setText(Html.fromHtml(simpleResult));
  }
}
```

Notre bouton contient désormais l'icône désirée (voir Figure 20.3).

Figure 20.3
L'application `ImagesDemo`.

Les ressources XML

Pour assembler des documents XML statiques avec une application, il suffit d'utiliser des ressources XML.

Placez simplement un fichier XML dans `res/xml/` et vous pourrez y accéder en appelant la méthode `getXml()` sur un objet `Resources` et en lui passant en paramètre un identifiant préfixé par `R.xml`, suivi du nom de base du fichier XML. Si une activité est fournie avec un fichier `words.xml`, par exemple, vous pouvez appeler `getResources().getXml(R.xml.words)` afin d'obtenir une instance de la classe `XmlPullParser`, qui se trouve dans l'espace de noms `org.xmlpull.v1`.

Un analyseur pull XML est piloté par les événements : il suffit d'appeler sa méthode `next()` pour obtenir l'événement suivant, qui peut être `START_TAG`, `END_TAG`, `END_DOCUMENT`, etc. Sur un événement `START_TAG`, vous pouvez accéder au nom du marqueur et à ses attributs ; un unique événement `TEXT` représente la concaténation de tous les nœuds qui sont les fils directs de cet élément. En itérant, en testant et en invoquant du code élément par élément, vous finissez par analyser tout le fichier.

Ce nouveau projet `Resources/XML` exige que vous placiez le fichier `words.xml` dans `res/xml` :

```
package com.commonsware.android.resources;

import android.app.Activity;
import android.os.Bundle;
import android.app.ListActivity;
```

```
import android.view.View;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.TextView;
import android.widget.Toast;
import java.io.InputStream;
import java.util.ArrayList;
import org.xmlpull.v1.XmlPullParser;
import org.xmlpull.v1.XmlPullParserException;

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

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

    try {
      XmlPullParser xpp=getResources().getXml(R.xml.words);

      while (xpp.getEventType()!=XmlPullParser.END_DOCUMENT) {
        if (xpp.getEventType()==XmlPullParser.START_TAG) {
          if (xpp.getName().equals("word")) {
            items.add(xpp.getAttributeValue(0));
          }
        }

        xpp.next();
      }
    }
    catch (Throwable t) {
      Toast
        .makeText(this, "Echec de la requete : "+ t.toString(), 4000)
        .show();
    }
```

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

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

Désormais, dans notre bloc `try...catch`, nous obtenons un `XmlPullParser` et nous bouclons jusqu'à la fin du document. Si l'événement courant est `START_TAG` et que le nom de l'élément soit `word` (`xpp.getName(). equals("word")`), nous récupérons le seul et unique attribut et nous l'ajoutons à la liste de mots du widget de sélection. Comme nous avons un contrôle total sur le fichier XML (puisque c'est nous qui l'avons créé), nous pouvons supposer qu'il n'y a exactement qu'un attribut – si nous n'en étions pas sûrs, nous pourrions compter leur nombre avec `getAttributeCount()` et obtenir leur nom avec `getAttributeName()` au lieu de supposer aveuglément que l'attribut d'indice 0 est celui auquel on pense.

Le résultat obtenu est présenté à la Figure 20.4.

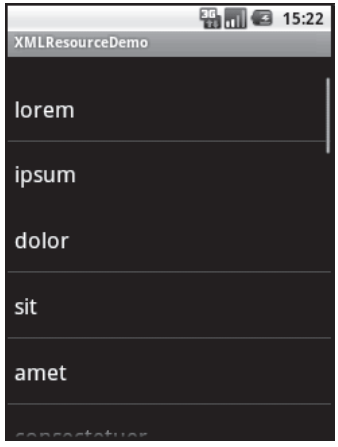


Figure 20.4
L'application `XMLResourceDemo`.

Valeurs diverses

Outre des ressources chaînes, vous pouvez placer dans le répertoire `res/values/` un ou plusieurs fichiers XML décrivant des ressources simples : des dimensions, des couleurs

et des tableaux. Dans les exemples précédents, nous avons déjà vu des utilisations des dimensions et des couleurs, qui étaient passées sous forme de chaînes simples aux appels de méthodes (`"10px"`, par exemple). Vous pouvez, bien sûr, configurer ces valeurs comme des objets Java constants et utiliser leurs noms symboliques, mais cela ne peut fonctionner que dans le code source Java, pas dans les fichiers de description XML. En les plaçant dans des fichiers de ressources XML, vous pouvez faire référence à ces valeurs à la fois dans du code Java et dans les fichiers de description ; en outre, elles sont ainsi centralisées au même endroit, ce qui facilite leur maintenance.

Les fichiers ressources XML ont pour racine l'élément `resources` ; tous les autres éléments sont des fils de cette racine.

Dimensions

Android utilise les dimensions en de nombreux endroits pour décrire des distances, comme la valeur de remplissage d'un widget. Bien que nous utilisions souvent le pixel comme unité de mesure (`10px` pour 10 pixels, par exemple), vous pouvez en choisir d'autres :

- `in` et `mm` indiquent, respectivement, des pouces et des millimètres, d'après la taille physique de l'écran.
- `pt` représente des points, c'est-à-dire 1/72e de pouce en termes typographiques, là aussi d'après la taille physique de l'écran.
- `dp` (*device-independant pixel*) et `sp` (s*cale-independant pixel*) indiquent des pixels indépendants du terminal – un pixel est égal à un dp pour un écran de 160 dpi, le facteur d'échelle reposant sur la densité de l'écran (les pixels indépendants de l'échelle tiennent également compte de la taille de la police choisie par l'utilisateur).

Pour encoder une dimension comme une ressource, ajoutez un élément `dimen` avec un attribut `name` nommant de façon unique cette ressource. Le contenu de cet élément est un texte représentant la valeur de la dimension :

```
<resources>
  <dimen name="fin">10px</dimen>
  <dimen name="epais">1in</dimen>
</resources>
```

Dans un fichier de description, les dimensions peuvent être référencées par `@dimen/`*`nom`*, où *`nom`* est le nom unique de la ressource (`fin` ou `epais`, dans l'exemple précédent). Dans le code Java, il suffit d'utiliser le nom unique préfixé par `R.dimen` (`Resources.getDimen(R.dimen.fin)`, par exemple).

Couleurs

Les couleurs Android s'expriment en valeurs RGB hexadécimales et peuvent éventuellement indiquer un canal alpha. Vous avez le choix entre des valeurs hexadécimales d'un seul caractère ou de deux caractères, ce qui donne donc quatre formats possibles :

- #RGB ;
- #ARGB ;
- #RRGGBB ;
- #AARRGGBB.

Ces valeurs sont les mêmes que dans les feuilles de style CSS (*Cascading Style Sheets*).

Vous pouvez, bien sûr, les placer comme des chaînes de caractères dans le code Java ou les fichiers de description. Cependant, il suffit d'ajouter des éléments `color` au fichier de ressource afin de les gérer comme des ressources. Ces éléments doivent également posséder un attribut `name` pour donner un nom unique à la couleur et contenir la valeur RGB :

```
<resources>
  <color name="jaune_orange">#FFD555</color>
  <color name="vert_foret">#005500</color>
  <color name="ambre_fonce">#8A3324</color>
</resources>
```

Dans un fichier de description, ces couleurs peuvent être désignées par `@color/`*`nom`*, où *`nom`* est le nom unique de la couleur (`ambre_fonce`, par exemple). En Java, préfixez ce nom unique par `R.color` (`Resources.getColor(R.color.vert_foret)`, par exemple).

Tableaux

Les ressources tableaux sont conçues pour contenir des listes de chaînes simples, comme une liste de civilités (M., Mme, Mlle, Dr, etc.).

Dans le fichier ressource, vous avez besoin d'un élément `string-array` ayant un attribut `name` pour donner un nom unique au tableau. Cet élément doit avoir autant de fils `item` qu'il y a d'éléments dans le tableau. Le contenu de chacun de ces fils est la valeur de l'entrée correspondante :

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

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

Dans le code Java, vous pouvez ensuite utiliser `Resources.getStringArray(R.array.`*`nom`*`)` pour obtenir un tableau de `String` contenant tous les éléments du tableau *`nom`* (`Resources.getStringArray(R.array.codes_aeroports)`, par exemple).

Gérer la différence

Un même ensemble de ressources peut ne pas convenir à toutes les situations d'utilisation de votre application. Un exemple évident est celui des ressources chaînes et la gestion de l'internationalisation (I18N) et de la localisation (L10N). N'utiliser que des chaînes d'une même langue fonctionne parfaitement – au moins pour le développeur – mais cela produit une application qui réduit le nombre de ses utilisateurs potentiels.

Les ressources peuvent également différer sur d'autres points :

- **L'orientation de l'écran.** L'écran est-il en mode portrait (vertical) ou en mode paysage (horizontal) ? Il peut également être carré, ce qui signifie qu'il n'a donc pas d'orientation particulière.
- **La taille de l'écran.** Combien a-t-il de pixels ? Vous devez en tenir compte pour accorder en conséquence la taille de vos ressources (petites ou grandes icônes, par exemple).
- **Écran tactile**. Si votre écran est tactile, est-il manipulable avec un stylet ou avec le doigt ?

- **Clavier.** De quel clavier dispose l'utilisateur (alphanumérique, numérique, les deux) ? Est-il toujours disponible ou est-ce une option ?
- **Autres dispositifs de saisie.** Le terminal a-t-il d'autres moyens de saisie, comme un pad directionnel ou une molette ?

Pour gérer toutes ces différences, Android utilise plusieurs répertoires de ressources dont les noms contiennent le critère concerné.

Supposons, par exemple, que vous vouliez fournir des textes anglais et français. Pour une application non traduite, vous placeriez normalement ces chaînes dans le fichier `res/values/strings.xml` mais, pour pouvoir reconnaître à la fois l'anglais et le français, vous devrez créer deux répertoires, `res/values-en` et `res/values-fr`, où les deux lettres qui suivent le tiret représentent la langue souhaitée selon le codage ISO-639-1[1]. Les chaînes anglaises seront donc placées dans le fichier `res/values-en/strings.xml` et les françaises, dans `res/values-fr/strings.xml`. Android choisira alors le bon fichier en fonction de la configuration du terminal de l'utilisateur.

Cela semble simple, n'est-ce pas ?

Les choses se compliquent lorsque vous avez besoin de plusieurs critères distincts pour vos ressources. Supposons, par exemple, que vous vouliez développer une application à la fois pour le G1 et deux autres terminaux fictifs. L'un d'eux (le Fictif Un) dispose d'un écran VGA généralement en mode paysage ("large"), d'un clavier alphanumérique toujours ouvert, d'un pad directionnel, mais pas d'écran tactile. L'autre (le Fictif Deux) a le même écran que le G1 ("normal"), un clavier numérique mais pas alphabétique, un pad directionnel mais pas d'écran tactile. Pour tirer parti de ces différences d'écrans et d'options de saisie, vous pourriez créer des fichiers de description différents :

- pour chaque combinaison de résolution et d'orientation ;
- pour les terminaux qui ont un écran tactile et ceux qui n'en ont pas ;
- pour les terminaux qui ont des claviers alphanumériques et ceux qui n'en ont pas.

Dans ces situations, toutes sortes de règles entrent en jeu :

- Les options de configuration (`-en`, par exemple) ont une certaine priorité et doivent apparaître dans cet ordre dans le nom du répertoire. La documentation d'Android[2] décrit l'ordre précis dans lequel ces options peuvent apparaître. Pour les besoins de notre exemple, l'orientation de l'écran doit précéder le type de l'écran (tactile ou non), qui doit lui-même précéder sa taille.

1. http://fr.wikipedia.org/wiki/Liste_des_codes_ISO_639-1.
2. http://code.google.com/android/devel/resources-i18n.html#AlternateResources.

- Il ne peut exister qu'une seule valeur par répertoire pour chaque catégorie d'option de configuration.
- Les options sont sensibles à la casse.

Pour notre scénario, nous aurions donc besoin, en théorie, des répertoires suivants :

- `res/layout-large-port-notouch-qwerty`
- `res/layout-normal-port-notouch-qwerty`
- `res/layout-large-port-notouch-12key`
- `res/layout-normal-port-notouch-12key`
- `res/layout-large-port-notouch-nokeys`
- `res/layout-normal-port-notouch-nokeys`
- `res/layout-large-port-stylus-qwerty`
- `res/layout-normal-port-stylus-qwerty`
- `res/layout-large-port-stylus-12key`
- `res/layout-normal-port-stylus-12key`
- `res/layout-large-port-stylus-nokeys`
- `res/layout-normal-port-stylus-nokeys`
- `res/layout-large-port-finger-qwerty`
- `res/layout-normal-port-finger-qwerty`
- `res/layout-large-port-finger-12key`
- `res/layout-normal-port-finger-12key`
- `res/layout-large-port-finger-nokeys`
- `res/layout-normal-port-finger-nokeys`
- `res/layout-large-land-notouch-qwerty`
- `res/layout-normal-land-notouch-qwerty`
- `res/layout-large-land-notouch-12key`
- `res/layout-normal-land-notouch-12key`
- `res/layout-large-land-notouch-nokeys`
- `res/layout-normal-land-notouch-nokeys`
- `res/layout-large-land-stylus-qwerty`
- `res/layout-normal-land-stylus-qwerty`
- `res/layout-large-land-stylus-12key`
- `res/layout-normal-land-stylus-12key`

- `res/layout-large-land-stylus-nokeys`
- `res/layout-normal-land-stylus-nokeys`
- `res/layout-large-land-finger-qwerty`
- `res/layout-normal-land-finger-qwerty`
- `res/layout-large-land-finger-12key`
- `res/layout-normal-land-finger-12key`
- `res/layout-large-land-finger-nokeys`
- `res/layout-normal-land-finger-nokeys`

Pas de panique ! Nous allons abréger cette liste dans un petit moment !

En réalité, beaucoup de ces fichiers de description seront identiques. Par exemple, nous voulons simplement que les descriptions des écrans tactiles soient différentes de celles des écrans non tactiles. Cependant, comme nous ne pouvons pas combiner les deux types d'écrans, nous devrons théoriquement avoir des répertoires distincts avec des contenus identiques pour les écrans manipulables avec le doigt (`finger`) et ceux manipulables avec un stylet (`stylus`).

Notez également que rien n'empêche d'avoir un répertoire avec un nom de base simple (`res/layout`). En réalité, c'est même sûrement préférable au cas où une nouvelle version d'Android introduirait d'autres options de configuration que vous n'avez pas prises en compte – disposer d'une description par défaut permettra alors à votre application de fonctionner sur ce nouveau terminal.

Nous pouvons maintenant "tricher" un peu en décodant les règles qu'utilise Android pour déterminer le "bon" répertoire des ressources parmi l'ensemble des candidats :

1. Premièrement, Android élimine les candidats invalides dans le contexte. Si la taille de l'écran du terminal est normale, par exemple, les répertoires `-larges` seront éliminés des candidats possibles car ils font spécifiquement appel à une autre taille.
2. Deuxièmement, Android compte le nombre de correspondances pour chaque répertoire et ne conserve que les répertoires qui en ont le plus.
3. Enfin, Android suit l'ordre de priorité des options – en d'autres termes, il parcourt le nom du répertoire de gauche à droite.

Nous pouvons donc nous ramener aux configurations suivantes :

- `res/layout-large-port-notouch-qwerty`
- `res/layout-port-notouch-qwerty`
- `res/layout-large-port-notouch`

- `res/layout-port-notouch`
- `res/layout-large-port-qwerty`
- `res/layout-port-qwerty`
- `res/layout-large-port`
- `res/layout-port`
- `res/layout-large-land-notouch-qwerty`
- `res/layout-land-notouch-qwerty`
- `res/layout-large-land-notouch`
- `res/layout-land-notouch`
- `res/layout-large-land-qwerty`
- `res/layout-land-qwerty`
- `res/layout-large-land`
- `res/layout-land`

Ici, nous tirons parti du fait que les correspondances spécifiques ont priorité sur les valeurs "non spécifiées". Ainsi, un terminal disposant d'un clavier alphanumérique choisira une ressource ayant `qwerty` dans son nom de répertoire plutôt qu'une ressource qui ne précise pas son type de clavier. Si l'on combine cet état de fait avec la règle "le plus grand nombre de correspondances l'emporte", nous voyons que `res/layout-port` ne correspondra qu'aux terminaux dotés d'écrans de taille normale, sans clavier alphanumérique et avec un écran tactile orienté en mode portrait.

Nous pourrions préciser tout cela encore un peu plus, pour ne couvrir que les terminaux que nous visons (le G1, Fictif Un et Fictif Deux) et en gardant `res/layout` comme description par défaut :

- `res/layout-large-port-notouch`
- `res/layout-port-notouch`
- `res/layout-large-land-notouch`
- `res/layout-land-notouch`
- `res/layout-land`
- `res/layout`

Ici, `-large` permet de différencier Fictif Un des deux autres, tandis que `notouch` distingue Fictif Deux du G1.

Nous rencontrerons à nouveau ces ressources au Chapitre 36, lorsque nous expliquerons comment gérer les tailles d'écrans différentes.

21

Utilisation des préférences

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

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

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

Obtenir ce que vous voulez

Pour accéder aux préférences, vous pouvez :

- appeler la méthode `getPreferences()` à partir de votre activité pour accéder à ses préférences spécifiques ;
- appeler la méthode `getSharedPreferences()` à partir de votre activité (ou d'un autre `Context` de l'application) pour accéder aux préférences de l'application ;
- appeler la méthode `getDefaultSharedPreferences()` d'un objet `PreferencesManager` pour accéder aux préférences partagées qui fonctionnent de concert avec le système des préférences globales d'Android.

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

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

Définir vos préférences

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

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

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

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

Un mot sur le framework

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

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

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

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

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

Laisser les utilisateurs choisir

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

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

```
package com.commonsware.android.prefs;

import android.app.Activity;
import android.os.Bundle;
import android.preference.PreferenceActivity;

public class EditPreferences extends PreferenceActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

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

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

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

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

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

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  menu.add(Menu.NONE, EDIT_ID, Menu.NONE, "Modifier Prefs")
      .setIcon(R.drawable.misc)
      .setAlphabeticShortcut('m');
```

```
    return(super.onCreateOptionsMenu(menu));
  }
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
      case EDIT_ID:
        startActivity(new Intent(this, EditPreferences.class));
        return(true);
    }
    return(super.onOptionsItemSelected(item));
  }
```

La Figure 21.1 montre l'interface de configuration des préférences de cette application.

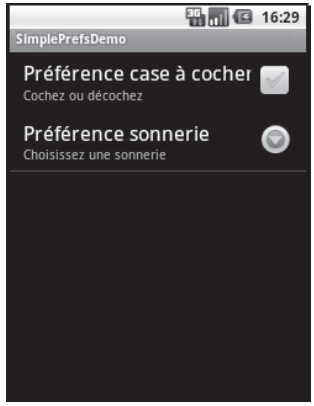


Figure 21.1
L'interface des préférences de `SimplePrefsDemo`.

La case à cocher peut être cochée ou décochée directement. Pour modifier la sonnerie, il suffit de cliquer sur son entrée dans les préférences pour voir apparaître une boîte de dialogue comme celle de la Figure 21.2.

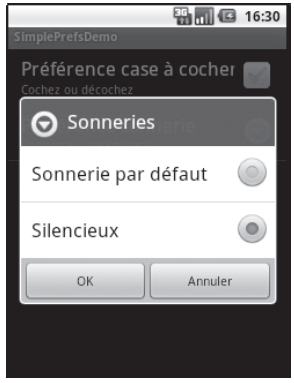


Figure 21.2
Choix d'une sonnerie à partir des préférences.

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

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

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

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

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

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

```
@Override
public void onResume() {
```

```
    super.onResume();
    SharedPreferences prefs=PreferenceManager
                            .getDefaultSharedPreferences(this);

    checkbox.setText(new Boolean(prefs
                            .getBoolean("checkbox", false))
                            .toString());
    ringtone.setText(prefs.getString("ringtone", "<unset>"));
  }
```

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

La Figure 21.3 montre le contenu de cette table après le choix de l'utilisateur.

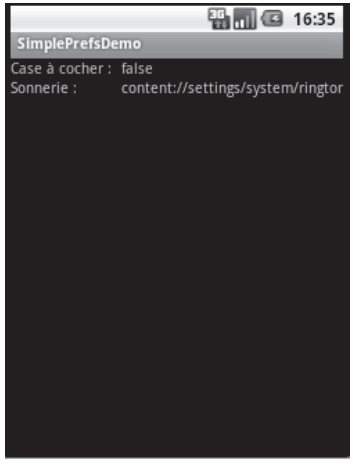


Figure 21.3
Liste des préférences de `SimplePrefsDemo`.

Ajouter un peu de structure

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

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

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

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

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

```
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android">
  <PreferenceCategory android:title="Préférences simples">
    <CheckBoxPreference
      android:key="@string/checkbox"
      android:title="Préférence case à cocher"
      android:summary="Cochez ou décochez"
    />
    <RingtonePreference
      android:key="@string/ringtone"
      android:title="Préférence sonnerie"
      android:showDefault="true"
      android:showSilent="true"
      android:summary="Choisissez une sonnerie"
    />
  </PreferenceCategory>
  <PreferenceCategory android:title="Détails supplémentaires">
    <PreferenceScreen
      android:key="detail"
      android:title="Détails"
      android:summary="Préférences supplémentaires dans une autre page">
      <CheckBoxPreference
        android:key="@string/checkbox2"
        android:title="Une autre case"
        android:summary="On ou Off. Peu importe."
      />
    </PreferenceScreen>
  </PreferenceCategory>
</PreferenceScreen>
```

Utilisé avec notre implémentation de `PreferenceActivity`, ce fichier XML produit une liste d'éléments comme ceux de la Figure 21.4, regroupés en catégories.

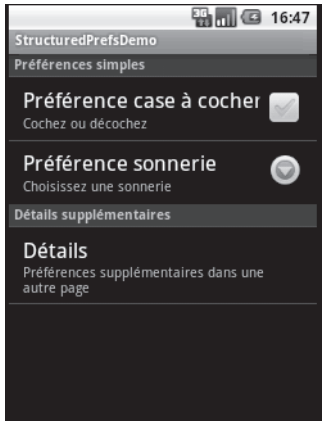


Figure 21.4
L'interface des préférences de `StructuredPrefsDemo`*, montrant les catégories et un marqueur d'écran.*

En touchant l'entrée du sous-écran, celui-ci s'affiche et montre les préférences qu'il contient (voir Figure 21.5).

Figure 21.5
Le sous-écran de préférences de `StructuredPrefsDemo`.

Boîtes de dialogue

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

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

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

```
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android">
  <PreferenceCategory android:title="Préférences simples">
    <CheckBoxPreference
      android:key="@string/checkbox"
      android:title="Préférence case à cocher"
      android:summary="Cochez ou décochez"
    />
    <RingtonePreference
      android:key="@string/ringtone"
      android:title="Préférence sonnerie"
      android:showDefault="true"
      android:showSilent="true"
      android:summary="Choisissez une sonnerie"
    />
  </PreferenceCategory>
  <PreferenceCategory android:title="Détails supplémentaires">
    <PreferenceScreen
      android:key="detail"
      android:title="Détails"
      android:summary="Préférences supplémentaires dans une autre page">
      <CheckBoxPreference
        android:key="@string/checkbox2"
        android:title="Autre case à cocher"
        android:summary="On ou Off. Peu importe."
      />
    </PreferenceScreen>
  </PreferenceCategory>
  <PreferenceCategory android:title="Préférences simples">
    <EditTextPreference
      android:key="@string/text"
      android:title="Dialogue de saisie d'un texte"
      android:summary="Cliquez pour ouvrir un champ de saisie"
      android:dialogTitle="Entrez un texte intéressant"
    />
    <ListPreference
      android:key="@string/list"
```

```
        android:title="Dialogue de choix"
        android:summary="Cliquez pour ouvrir une liste de choix"
        android:entries="@array/villes"
        android:entryValues="@array/codes_aeroports"
        android:dialogTitle="Choisissez une ville" />
    </PreferenceCategory>
</PreferenceScreen>
```

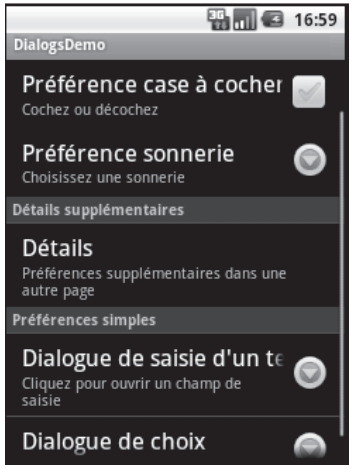
Pour le champ de texte (`EditTextPreference`), outre le titre et le résumé de la préférence elle-même, nous donnons également un titre à la boîte de dialogue.

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

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string-array name="villes">
    <item>Philadelphia</item>
    <item>Pittsburgh</item>
    <item>Allentown/Bethlehem</item>
    <item>Erie</item>
    <item>Reading</item>
    <item>Scranton</item>
    <item>Lancaster</item>
    <item>Altoona</item>
    <item>Harrisburg</item>
  </string-array>
  <string-array name="codes_aeroports">
    <item>PHL</item>
    <item>PIT</item>
    <item>ABE</item>
    <item>ERI</item>
    <item>RDG</item>
    <item>AVP</item>
    <item>LNS</item>
    <item>AOO</item>
    <item>MDT</item>
  </string-array>
</resources>
```

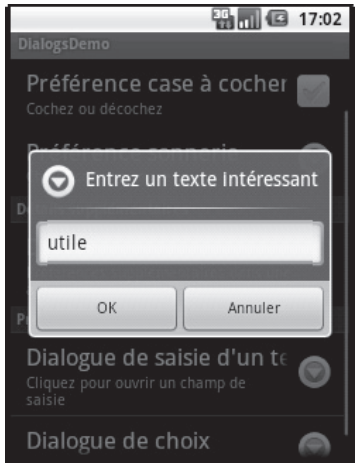
Comme le montre la Figure 21.6, les préférences comprennent désormais une catégorie supplémentaire contenant deux nouvelles entrées.

Figure 21.6
L'écran des préférences de `DialogsDemo`.



Toucher l'entrée Dialogue de saisie d'un texte provoque l'affichage d'un... dialogue de saisie d'un texte – ici, il est prérempli avec la valeur courante de la préférence (voir Figure 21.7).

Figure 21.7
Modification d'une préférence avec un champ de saisie.



Toucher l'entrée Dialogue de choix affiche... une liste de choix sous forme de boîte de dialogue présentant les noms des villes (voir Figure 21.8).

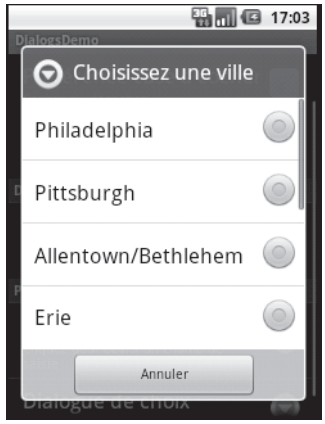


Figure 21.8
Modification d'une préférence avec une liste.

22

Accès et gestion des bases de données locales

SQLite[1] est une base de données très appréciée car elle fournit une interface SQL tout en offrant une empreinte mémoire très réduite et une rapidité de traitement satisfaisante. En outre, elle appartient au domaine public et tout le monde peut donc l'utiliser. De nombreuses sociétés (Adobe, Apple, Google, Sun, Symbian) et plusieurs projets open-source (Mozilla, PHP, Python) fournissent désormais des produits intégrant SQLite.

SQLite étant intégré au moteur d'exécution d'Android, toute application peut créer des bases de données SQLite. Ce SGBD disposant d'une interface SQL, son utilisation est assez simple pour quiconque a une expérience avec d'autres SGBDR. Cependant, son API native n'est pas JDBC, qui, d'ailleurs, serait trop lourd pour les terminaux limités en mémoire comme les téléphones. Par conséquent, les programmeurs Android doivent apprendre une nouvelle API ; comme nous allons le voir, ce n'est pas très difficile.

1. **http://www.sqlite.org**.

Ce chapitre présente les bases de l'utilisation de SQLite dans le contexte du développement Android. Il ne prétend absolument pas être une présentation exhaustive de ce SGBDR : pour plus de renseignements et pour savoir comment l'utiliser dans d'autres environnements qu'Android, nous vous conseillons l'ouvrage de Mike Owens, *The Definitive Guide to SQLite*[1] (Apress, 2006).

Exemple de base de données

L'essentiel du code de l'exemple présenté dans ce chapitre provient de l'application `Database/Constants`, qui, comme le montre la Figure 22.1, affiche la liste de constantes définies par la classe `SensorManager` d'Android.

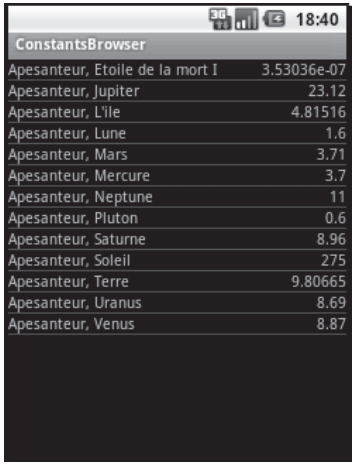


Figure 22.1
L'application `Constants`*, juste après son lancement.*

Vous pouvez ouvrir un menu pour ajouter une nouvelle constante : comme le montre la Figure 22.2, une boîte de dialogue s'affiche alors pour vous permettre de saisir son nom et sa valeur.

La constante saisie est ajoutée à la liste. Un touché prolongé sur une constante de la liste affiche un menu contextuel comprenant une option Supprimer qui, après confirmation, supprime la constante concernée.

1. **http://www.amazon.com/Definitive-Guide-SQLite/dp/1590596730**.

Figure 22.2
L'application `Constants`*, juste après son lancement.*

Ces données sont bien entendu stockées dans une base de données SQLite.

Présentation rapide de SQLite

SQLite, comme son nom l'indique, utilise un dialecte de SQL pour effectuer des requêtes (`SELECT`), des manipulations de données (`INSERT`, etc.) et des définitions de données (`CREATE TABLE`, etc.). À certains moments, SQLite s'écarte du standard SQL-92, comme la plupart des autres SGBDR, d'ailleurs. La bonne nouvelle est que SQLite est si efficace en terme de mémoire que le moteur d'exécution d'Android peut l'inclure dans son intégralité : vous n'êtes donc pas obligé de vous contenter d'un sous-ensemble de ses fonctionnalités pour gagner de la place.

La plus grosse différence avec les autres SGBDR concerne principalement le typage des données. Bien que vous puissiez préciser les types des colonnes dans une instruction `CREATE TABLE`, SQLite ne les utilisera qu'à titre indicatif.Vous pouvez donc mettre les données que vous voulez dans les colonnes que vous souhaitez. Vous voulez placer une chaîne dans une colonne `INTEGER` ? Pas de problème ! Et *vice versa* ? Cela marche aussi ! C'est ce que SQLite appelle "typage manifeste" ; il est décrit de la façon suivante dans sa documentation[1] :

Avec le typage manifeste, le type d'une donnée est une propriété de la valeur elle-même, pas de la colonne dans laquelle la valeur est stockée. SQLite permet donc de stocker une valeur de n'importe quel type dans n'importe quelle colonne, quel que soit le type déclaré de cette colonne.

---

1. **http://www.sqlite.org/different.html**.

Certaines fonctionnalités standard de SQL ne sont pas reconnues par SQLite, notamment les contraintes de clé étrangère, les transactions imbriquées, les jointures externes et certaines variantes de `ALTER TABLE`.

Ces remarques mises à part, vous disposez d'un SGBDR complet, avec des triggers, des transactions, etc. Les instructions SQL de base comme `SELECT` fonctionnent exactement comme vous êtes en droit de l'attendre.

Si vous êtes habitué à travailler avec un gros SGBDR comme Oracle, vous pourriez considérer que SQLite est un "jouet", mais n'oubliez pas que ces deux systèmes ont été conçus pour résoudre des problèmes différents et que vous n'êtes pas près de voir une installation complète d'Oracle sur un téléphone.

Commencer par le début

Android ne fournit aucune base de données de son propre chef. Si vous voulez utiliser SQLite, vous devez créer votre propre base, puis la remplir avec vos tables, vos index et vos données.

Pour créer et ouvrir une base de données, la meilleure solution consiste à créer une sous-classe de `SQLiteOpenHelper`. Cette classe enveloppe tout ce qui est nécessaire à la création et à la mise à jour d'une base, selon vos spécifications et les besoins de votre application. Cette sous-classe aura besoin de trois méthodes :

- Un constructeur qui appelle celui de sa classe parente et qui prend en paramètre le `Context` (une `Activity`), le nom de la base de données, une éventuelle fabrique de curseur (le plus souvent, ce paramètre vaudra `null`) et un entier représentant la version du schéma de la base.
- `onCreate()`, à laquelle vous passerez l'objet `SQLiteDatabase` que vous devrez remplir avec les tables et les données initiales que vous souhaitez.
- `onUpgrade()`, à laquelle vous passerez un objet `SQLiteDatabase` ainsi que l'ancien et le nouveau numéro de version afin de pouvoir convertir au mieux une base de données d'un schéma vers un autre. Pour convertir une base d'un ancien schéma à un nouveau, l'approche la plus simple consiste à supprimer les anciennes tables et à en créer de nouvelles.

Voici, par exemple, la classe `DatabaseHelper` de `Database/Constants`, qui crée une table dans `onCreate()` et lui ajoute un certain nombre de lignes. Dans `onUpgrade()`, elle triche un peu en supprimant la table existante et en appelant à nouveau `onCreate()` :

```
package com.commonsware.android.constants;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteDatabase;
import android.hardware.SensorManager;

public class DatabaseHelper extends SQLiteOpenHelper {
  private static final String DATABASE_NAME="db";
  public static final String TITLE="title";
  public static final String VALUE="value";

  public DatabaseHelper(Context context) {
    super(context, DATABASE_NAME, null, 1);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    db.execSQL("CREATE TABLE constants (_id INTEGER <FL>
                PRIMARY KEY AUTOINCREMENT, title TEXT, value REAL);");

    ContentValues cv=new ContentValues();

    cv.put(TITLE, "Apesanteur, Etoile de la mort I");
    cv.put(VALUE, SensorManager.GRAVITY_DEATH_STAR_I);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Apesanteur, Terre");
    cv.put(VALUE, SensorManager.GRAVITY_EARTH);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Apesanteur, Jupiter");
    cv.put(VALUE, SensorManager.GRAVITY_JUPITER);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Apesanteur, Mars");
    cv.put(VALUE, SensorManager.GRAVITY_MARS);
    db.insert("constants", TITLE, cv);
```

```
    cv.put(TITLE, "Apesanteur, Mercure");
    cv.put(VALUE, SensorManager.GRAVITY_MERCURY);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Apesanteur, Lune");
    cv.put(VALUE, SensorManager.GRAVITY_MOON);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Apesanteur, Neptune");
    cv.put(VALUE, SensorManager.GRAVITY_NEPTUNE);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Apesanteur, Pluton");
    cv.put(VALUE, SensorManager.GRAVITY_PLUTO);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Apesanteur, Saturne");
    cv.put(VALUE, SensorManager.GRAVITY_SATURN);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Apesanteur, Soleil");
    cv.put(VALUE, SensorManager.GRAVITY_SUN);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Apesanteur, L'ile");
    cv.put(VALUE, SensorManager.GRAVITY_THE_ISLAND);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Apesanteur, Uranus");
    cv.put(VALUE, SensorManager.GRAVITY_URANUS);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Apesanteur, Venus");
    cv.put(VALUE, SensorManager.GRAVITY_VENUS);
    db.insert("constants", TITLE, cv);
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    android.util.Log.w("Constants",
```

```
                "Maj de la base, suppression de toutes les anciennes donnees");
        db.execSQL("DROP TABLE IF EXISTS constants");
        onCreate(db);
      }
    }
```

Pour utiliser cette sous-classe de `SQLiteOpenHelper`, créez une instance et demandez-lui d'appeler `getReadableDatabase()` ou `getWriteableDatabase()` selon que vous vouliez ou non modifier son contenu. Notre activité `ConstantsBrowser`, par exemple, ouvre la base de données dans `onCreate()` :

```
db=(new DatabaseHelper(getContext())).getWritableDatabase();
```

Cet appel renverra une instance de `SQLiteDatabase` qui vous servira ensuite à interroger ou à modifier la base de données.

Lorsque vous avez fini de travailler sur cette base (lorsque l'activité est fermée, par exemple), il suffit d'appeler la méthode `close()` de cette instance pour libérer votre connexion.

Mettre la table

Pour créer des tables et des index, vous devez appeler la méthode `execSQL()` de l'objet `SQLiteDatabase` en lui passant l'instruction du LDD (*langage de définition des données*) que vous voulez exécuter. En cas d'erreur, cette méthode renvoie `null`.

La méthode `onCreate()` de `DatabaseHelper`, par exemple, utilise le code suivant :

```
db.execSQL("CREATE TABLE
constants (_id INTEGER PRIMARY KEY AUTOINCREMENT,<FL>
                                    title TEXT, value REAL);");
```

Cet appel crée une table `constants` avec une colonne de clé primaire `_id` qui est un entier incrémenté automatiquement (SQLite lui affectera une valeur pour vous lorsque vous insérerez les lignes). Cette table contient également deux colonnes de données : `title` (un texte) et `value` (un nombre réel). SQLite créera automatiquement un index sur la colonne de clé primaire – si vous le souhaitez, vous pouvez en ajouter d'autres à l'aide d'instructions `CREATE INDEX`.

Le plus souvent, vous créerez les tables et les index dès la création de la base de données ou, éventuellement, lorsqu'elle devra être mise à jour suite à une nouvelle version de votre application. Si les schémas des tables ne changent pas, les tables et les index n'ont pas besoin d'être supprimés mais, si vous devez le faire, il suffit d'utiliser `execSQL()` afin d'exécuter les instructions `DROP INDEX` et `DROP TABLE`.

Ajouter des données

Lorsque l'on crée une base de données et une ou plusieurs tables, c'est généralement pour y placer des données. Pour ce faire, il existe principalement deux approches :

- Utiliser `execSQL()`, comme vous l'avez fait pour créer les tables. Cette méthode permet en effet d'exécuter n'importe quelle instruction SQL qui ne renvoie pas de résultat, ce qui est le cas d'`INSERT`, `UPDATE`, `DELETE`, etc.
- Utiliser `insert()`, `update()` et `delete()` sur l'objet `SQLiteDatabase`. Ces méthodes "builder" prennent en paramètre une instruction SQL découpée en plusieurs morceaux. Elles utilisent des objets `ContentValues` qui implémentent une interface ressemblant à `Map` mais avec des méthodes supplémentaires pour prendre en compte les types de SQLite : outre `get()`, qui permet de récupérer une valeur par sa clé, vous disposez également de `getAsInteger()`, `getAsString()`, etc.

Voici, par exemple, comment insérer une ligne dans la table `constants` avec `insert()` :

```
private void processAdd(DialogWrapper wrapper) {
    ContentValues values=new ContentValues(2);

    values.put("title", wrapper.getTitle());
    values.put("value", wrapper.getValue());

    db.insert("constants", "title", values);
    constantsCursor.requery();
  }
```

La méthode `insert()` prend en paramètre le nom de la table, celui d'une colonne pour l'*astuce de la colonne nulle* et un objet `ContentValues` contenant les valeurs que vous voulez placer dans cette ligne. L'astuce de la colonne nulle est utilisée dans le cas où l'instance de `ContentValues` est vide – la colonne indiquée pour cette astuce recevra alors explicitement la valeur NULL dans l'instruction INSERT produite par `insert()`.

La méthode `update()` prend en paramètre le nom de la table, un objet `ContentValues` contenant les colonnes et leurs nouvelles valeurs et, éventuellement, une clause `WHERE` et une liste de paramètres qui remplaceront les marqueurs `?` présents dans celle-ci. `update()` n'autorisant que des valeurs fixes pour mettre à jour les colonnes, vous devrez utiliser `execSQL()` si vous souhaitez affecter des résultats calculés.

La clause `WHERE` et la liste de paramètres fonctionnent comme les paramètres positionnels qui existent également dans d'autres API de SQL.

La méthode `delete()` fonctionne comme `update()` : elle prend en paramètre le nom de la table et, éventuellement, une clause `WHERE` et une liste des paramètres positionnels pour cette clause. Voici, par exemple, comment nous supprimons la ligne d'`_ID` indiqué de la table `constants` :

```
private void processDelete(long rowId) {
    String[] args={String.valueOf(rowId)};

    db.delete("constants", "_ID=?", args);
    constantsCursor.requery();
  }
```

Le retour de vos requêtes

Comme pour `INSERT`, `UPDATE` et `DELETE`, vous pouvez utiliser plusieurs approches pour récupérer les données d'une base SQLite avec SELECT :

- `rawQuery()` permet d'exécuter directement une instruction `SELECT`.
- `query()` permet de construire une requête à partir de ses différentes composantes.

Un sujet de confusion classique est la classe `SQLiteQueryBuilder` et le problème des curseurs et de leurs fabriques : c'est la raison pour laquelle nous allons les présenter séparément.

Requêtes brutes

La solution la plus simple, au moins du point de vue de l'API, consiste à utiliser `rawQuery()` en lui passant simplement la requête `SELECT`. Cette dernière peut contenir des paramètres positionnels qui seront remplacés par les éléments du tableau passé en second paramètre. `Voici un exemple :`

```
constantsCursor=db.rawQuery("SELECT _ID, title, value "+
                            "FROM constants ORDER BY title",
                             null);La valeur renvoyée est un Cursor qui dispose
de méthodes permettant de parcourir le résultat (voir la section "Utilisation
des curseurs").
```

Si vos requêtes sont bien intégrées à votre application, c'est une approche très simple. En revanche, elle se complique lorsqu'une requête comprend des parties dynamiques que les paramètres positionnels ne peuvent plus gérer. Si l'ensemble de colonnes que vous voulez récupérer n'est pas connu au moment de la compilation, par exemple, concaténer les noms des colonnes pour former une liste délimitée par des virgules peut être ennuyeux – c'est là que `query()` entre en jeu.

Requêtes normales

La méthode `query()` prend en paramètre les parties d'une instruction `SELECT` afin de construire la requête. Ces différentes composantes apparaissent dans l'ordre suivant dans la liste des paramètres :

- le nom de la table interrogée ;
- la liste des colonnes à récupérer ;
- la clause `WHERE`, qui peut contenir des paramètres positionnels ;
- la liste des valeurs à substituer à ces paramètres positionnels ;
- une éventuelle clause `GROUP BY` ;
- une éventuelle clause `ORDER BY` ;
- une éventuelle clause `HAVING`.

À part le nom de la table, ces paramètres peuvent valoir `null` lorsqu'ils ne sont pas nécessaires :

```
String[] colonnes={"ID", "inventory"};
String[] params={"snicklefritz"};
Cursor result=db.query("widgets", colonnes, "name=?",
                       params, null, null, null);
```

Utilisation des "builders"

Une autre possibilité consiste à utiliser `SQLiteQueryBuilder`, qui offre bien plus de possibilités pour construire les requêtes complexes, notamment celles qui impliquent d'unir les résultats de plusieurs sous-requêtes, par exemple.

En outre, l'interface `SQLiteQueryBuilder` s'accorde parfaitement avec l'interface `ContentProvider` pour exécuter les requêtes. Un patron de conception classique pour l'implémentation de la méthode `query()` de votre fournisseur de contenu consiste donc à créer un objet `SQLiteQueryBuilder`, à lui fournir certaines valeurs par défaut, puis à lui faire construire (et, éventuellement, exécuter) la requête complète en combinant ces valeurs par défaut avec celles qui ont été passées au fournisseur de contenu lors de la demande de requête.

Voici, par exemple, un extrait de code d'un fournisseur de contenu utilisant `SQLite QueryBuilder` :

```
@Override
public Cursor query(Uri url, String[] projection, String selection,
```

```
                    String[] selectionArgs, String sort) {
  SQLiteQueryBuilder qb=new SQLiteQueryBuilder();
  qb.setTables(getTableName());
  if (isCollectionUri(url)) {
    qb.setProjectionMap(getDefaultProjection());
  }
  else {
    qb.appendWhere(getIdColumnName()+"="+url.getPathSegments().get(1));
  }
  String orderBy;
  if (TextUtils.isEmpty(sort)) {
    orderBy=getDefaultSortOrder();
  } else {
    orderBy=sort;
  }
  Cursor c=qb.query(db, projection, selection, selectionArgs,
                    null, null, orderBy);
  c.setNotificationUri(getContext().getContentResolver(), url);
  return c;
}
```

Les fournisseurs de contenu (*content provider*) seront expliqués en détail aux Chapitres 26 et 27. En attendant, nous pouvons nous contenter de remarquer que :

- Nous construisons un objet `SQLiteQueryBuilder`.
- Nous lui indiquons la table concernée par la requête avec `setTables(getTableName())`.
- Soit nous lui indiquons l'ensemble de colonnes à renvoyer par défaut (avec `setProjectionMap()`), soit nous lui donnons une partie de clause WHERE afin d'identifier une ligne précise de la table à partir d'un identifiant extrait de l'URI fournie à l'appel de `query()` (avec `appendWhere()`).
- Enfin, nous lui demandons d'exécuter la requête en mélangeant les valeurs de départ avec celles fournies à `query()` (`qb.query(db, projection, selection, selectionArgs, null, null, orderBy)`).

Au lieu de faire exécuter directement la requête par l'objet `SQLiteQueryBuilder`, nous aurions pu appeler `buildQuery()` pour la produire et renvoyer l'instruction `SELECT` dont nous avions besoin ; nous aurions alors pu l'exécuter nous-mêmes.

Utilisation des curseurs

Quelle que soit la façon dont vous exécutez la requête, vous obtiendrez un `Cursor` en retour. Il s'agit de la version Android/SQLite des curseurs de bases de données, un concept utilisé par de nombreux SGBDR. Avec ce curseur, vous pouvez effectuer les opérations suivantes :

- connaître le nombre de lignes du résultat grâce à `getCount()` ;
- parcourir les lignes du résultat avec `moveToFirst()`, `moveToNext()` et `isAfterLast()` ;connaître les noms des colonnes avec `getColumnNames()`, les convertir en numéros de colonnes grâce à `getColumnIndex()` et obtenir la valeur d'une colonne donnée de la ligne courante *via* des méthodes comme `getString()`, `getInt()`, etc. ;
- exécuter à nouveau la requête qui a créé le curseur, avec `requery()` ;
- libérer les ressources occupées par le curseur avec `close()`.

Voici, par exemple, comment parcourir les entrées d'une table `widgets` :

```
Cursor result=
  db.rawQuery("SELECT ID, name, inventory FROM widgets");

result.moveToFirst();

while (!result.isAfterLast()) {
  int id=result.getInt(0);
  String name=result.getString(1);
  int inventory=result.getInt(2);

  // Faire quelque chose de ces valeurs...

  result.moveToNext();
}

result.close();
```

Vous pouvez également encapsuler un `Cursor` dans un `SimpleCursorAdapter` ou une autre implémentation, puis passer l'adaptateur résultant à une `ListView` ou à un autre widget de sélection. Après avoir récupéré la liste des constantes triées, par exemple, nous plaçons chacune d'elles dans la `ListView` de l'activité `ConstantsBrowser` en quelques lignes de code :

```
ListAdapter adapter=new SimpleCursorAdapter(this,
                         R.layout.row, constantsCursor,
                         new String[] {"title", "value"},
                         new int[] {R.id.title, R.id.value});

setListAdapter(adapter);
```

Dans certains cas, vous pouvez vouloir utiliser votre propre sous-classe de `Cursor` *plutôt que l'implémentation de base fournie par Android. Dans ces situations, vous pouvez vous servir des méthodes* `queryWithFactory()` *et* `rawQueryWithFactory()`*, qui prennent toutes les deux en paramètre une instance de* `SQLiteDatabase.CursorFactory`*. Cette fabrique, comme l'on pourrait s'y attendre, est responsable de la création de nouveaux curseurs via son implémentation de* `newCursor()`.

Des données, des données, encore des données

Si vous avez l'habitude de développer avec d'autres SGBDR, vous avez sûrement aussi utilisé des outils permettant d'inspecter et de manipuler le contenu de la base de données et qui vont au-delà de l'API. Avec l'émulateur d'Android, vous avez également deux possibilités.

Premièrement, l'émulateur est censé fournir le programme `sqlite3`, accessible *via* la commande `adb shell`. Lorsque vous avez lancé cette commande, tapez simplement `sqlite3` suivi du chemin vers le fichier de votre base de données, qui est généralement de la forme :

```
/data/data/votre.paquetage.app/databases/nom_base
```

Ici, *`votre.paquetage.app`* est le paquetage Java de l'application (`com.commonsware.android`, par exemple) et *`nom_base`* est le nom de la base de données, tel qu'il est fourni à `createDatabase()`.

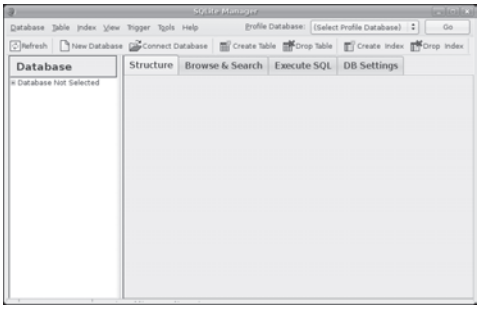
Le programme `sqlite3` fonctionne bien et, si vous avez l'habitude de manipuler vos tables à partir de la console, il vous sera très utile. Si vous préférez disposer d'une interface un peu plus conviviale, vous pouvez copier la base de données SQLite du terminal sur votre machine de développement, puis utiliser un client graphique pour SQLite. Cependant, n'oubliez pas que vous travaillez alors sur une copie de la base : si vous voulez répercuter les modifications sur le terminal, vous devrez retransférer cette base sur celui-ci.

Pour récupérer la base stockée sur le terminal, utilisez la commande `adb pull` (ou le gestionnaire de fichiers du *Dalvik Debug Monitor Service*, que nous présenterons au Chapitre 35) en lui fournissant le chemin de la base sur le terminal et celui de la destination sur votre machine. Pour faire l'inverse, utilisez la commande `adb push` en lui indiquant le chemin de cette base sur votre machine et le chemin de destination sur le terminal.

L'extension SQLite Manager[1] pour Firefox est l'un des clients SQLite les plus accessibles (voir Figure 22.3), car elle est disponible sur toutes les plates-formes.

1. **https://addons.mozilla.org/en-US/firefox/addon/5817**.

Figure 22.3
L'extension SQLite Manager de Firefox.



Vous trouverez également d'autres clients[1] sur le site web de SQLite[2].

1. **http://www.sqlite.org/cvstrac/wiki?p=SqliteTools**.
2. **http://www.sqlite.org**.

23

Accès aux fichiers

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

Allons-y !

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

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

si vous placez un fichier words.xml dans res/raw, son identifiant dans le code Java sera R.raw.words.

Comme vous ne pouvez obtenir qu'un InputStream, vous n'avez aucun moyen de modifier ce fichier : cette approche n'est donc vraiment utile que pour lire des données statiques. En outre, comme elles ne changeront pas jusqu'à ce que l'utilisateur installe une nouvelle version de votre paquetage, ces données doivent être valides pour une certaine période ou vous devrez fournir un moyen de les mettre à jour. Le moyen le plus simple de régler ce problème consiste à utiliser ces données pour construire une autre forme de stockage qui, elle, sera modifiable (une base de données, par exemple), mais cela impose d'avoir deux copies des données.

Une autre solution consiste à les conserver telles quelles et à placer les modifications dans un autre fichier ou dans une base de données, puis à les fusionner lorsque vous avez besoin d'une vue complète de ces informations. Si votre application fournit une liste d'URL, par exemple, vous pourriez gérer un deuxième fichier pour stocker les URL ajoutées par l'utilisateur ou pour référencer celles qu'il a supprimées.

Le projet Files/Static reprend l'exemple ListViewDemo du Chapitre 7 en utilisant cette fois-ci un fichier XML à la place d'un tableau défini directement dans le programme. Le fichier de description XML est identique dans les deux cas :

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

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

```
<words>
  <word value="lorem" />
  <word value="ipsum" />
```

```
  <word value="dolor" />
  <word value="sit" />
  <word value="amet" />
  <word value="consectetuer" />
  <word value="adipiscing" />
  <word value="elit" />
  <word value="morbi" />
  <word value="vel" />
  <word value="ligula" />
  <word value="vitae" />
  <word value="arcu" />
  <word value="aliquet" />
  <word value="mollis" />
  <word value="etiam" />
  <word value="vel" />
  <word value="erat" />
  <word value="placerat" />
  <word value="ante" />
  <word value="porttitor" />
  <word value="sodales" />
  <word value="pellentesque" />
  <word value="augue" />
  <word value="purus" />
</words>
```

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

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

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

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

    try {
```

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

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

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

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

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

Les différences entre cet exemple et celui du Chapitre 7 se situent essentiellement dans le corps de la méthode `onCreate()`. Ici, on obtient un `InputStream` pour le fichier XML en appelant `getResources().openRawResource(R.raw.words)`, puis on se sert des fonctionnalités d'analyse XML prédéfinies pour transformer le fichier en document DOM, en extraire les éléments `word` et placer les valeurs de leurs attributs `value` dans un objet `ArrayList` qui sera utilisé par l'`ArrayAdapter`.

Comme le montre la Figure 23.1, le résultat de l'activité est identique à celui l'exemple du Chapitre 7 car la liste de mots est la même.

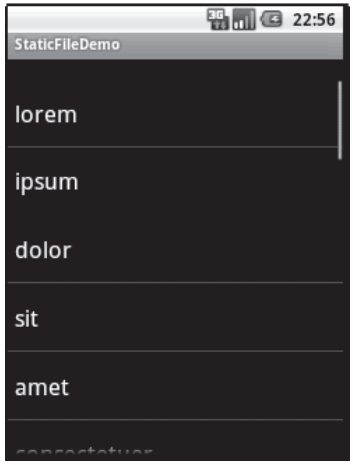


Figure 23.1
L'application `StaticFileDemo`.

Comme nous l'avons vu au Chapitre 20, il existe évidemment des moyens encore plus simples d'utiliser les fichiers XML contenus dans le paquetage d'une application : utiliser une ressource XML, par exemple. Cependant, bien que cet exemple utilise XML, le fichier aurait pu simplement contenir un mot par ligne ou utiliser un format non reconnu nativement par le système de ressources d'Android.

Lire et écrire

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

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

Deux applications qui essaient de lire en même temps un fichier `notes.txt` *via* `openFileInput()` accéderont chacune à leur propre édition du fichier. Si vous voulez qu'un même fichier soit accessible à partir de plusieurs endroits, vous devrez sûrement créer un *fournisseur de contenu*, comme on l'explique au Chapitre 27.

Notez également qu'`openFileInput()` et `openFileOutput()` ne prennent pas en paramètre des chemins d'accès (`chemin/vers/fichier.txt`, par exemple), mais uniquement des noms de fichiers.

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

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

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

```
package com.commonsware.android.files;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;
import java.io.BufferedReader;
import java.io.File;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
```

```
public class ReadWriteFileDemo extends Activity {
  private final static String NOTES="notes.txt";
  private EditText editor;

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

    Button btn=(Button)findViewById(R.id.close);

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

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

    try {
      InputStream in=openFileInput(NOTES);

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

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

        in.close();
        editor.setText(buf.toString());
      }
    }
    catch (java.io.FileNotFoundException e) {
      // OK, nous ne l'avons probablement pas encore créé
    }
    catch (Throwable t) {
```

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

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

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

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

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

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

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

Le résultat est un bloc-notes persistant ayant l'aspect présenté par les Figures 23.2 et 23.3. Tout ce qui est tapé le restera jusqu'à sa suppression ; le texte survivra à la fermeture de l'activité, à l'extinction du téléphone et aux autres situations similaires.

Figure 23.2
L'application `ReadWrite-FileDemo`*, après son lancement.*

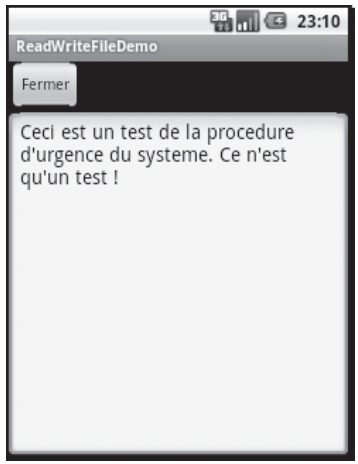


Figure 23.3
L'application `ReadWrite-FileDemo`*, après avoir saisi du texte.*

Vous pouvez également lire et écrire des fichiers sur la zone de stockage externe (c'est-à-dire la carte SD) en vous servant d'`Environment.getExternalStorageDirectory()` afin d'obtenir un objet `File` situé à la racine de la carte SD. À partir d'Android 1.6, vous devrez également posséder les permissions pour accéder à cette zone (`WRITE_EXTERNAL_STORAGE`, par exemple). Les permissions sont présentées au Chapitre 28.

N'oubliez pas que la zone de stockage externe est accessible à toutes les applications, alors qu'`openFileInput()` et `openFileOutput()` concernent la zone privée à l'application.

24

Tirer le meilleur parti des bibliothèques Java

Java a autant de bibliothèques tierces que les autres langages de programmation modernes, si ce n'est plus. Quand nous parlons de "bibliothèques tierces", nous faisons référence ici aux innombrables JAR que vous pouvez inclure dans une application Java, quelle qu'elle soit : cela concerne tout ce que le SDK Java ne fournit pas lui-même.

Dans le cas d'Android, le cœur de la machine virtuelle Dalvik n'est pas exactement Java et ce que fournit son SDK n'est pas la même chose qu'un SDK Java traditionnel. Ceci étant dit, de nombreuses bibliothèques Java fournissent les fonctionnalités dont ne dispose pas Android et peuvent donc vous être utiles (pour autant que vous puissiez les utiliser avec la variante Java d'Android).

Ce chapitre explique ce qu'il faut faire pour tirer parti de ces bibliothèques et décrit les limites de l'intégration du code tiers dans une application Android.

Limites extérieures

Tout le code Java existant ne fonctionne évidemment pas avec Android. Un certain nombre de facteurs doivent être pris en compte :

- **API pour la plate-forme.** Est-ce que le code suppose que vous utilisez une JVM plus récente que celle sur laquelle repose Android ou suppose-t-il l'existence d'une API Java fournie avec J2SE mais qui n'existe pas dans Android, comme Swing ?
- **Taille.** Le code Java conçu pour être utilisé sur les machines de bureau ou les serveurs ne se soucie pas beaucoup de l'espace disque, ni de la taille mémoire. Android, évidemment, manque des deux. L'utilisation de code tiers, notamment lorsqu'il est déjà assemblé sous forme de JAR, peut faire gonfler la taille de votre application.
- **Performances.** Est-ce que le code Java suppose un CPU beaucoup plus puissant que ceux que vous pouvez trouver sur la plupart des terminaux Android ? Ce n'est pas parce qu'un ordinateur de bureau peut l'exécuter sans problème qu'un téléphone mobile moyen pourra faire de même.
- **Interface.** Est-ce que le code Java suppose une interface en mode console, ou s'agit-il d'une API que vous pouvez envelopper dans votre propre interface ?

Une astuce pour régler quelques-uns de ces problèmes consiste à utiliser du code Java open-source et à le modifier pour l'adapter à Android. Si, par exemple, vous n'utilisez que 10 % d'une bibliothèque tierce, il est peut-être plus intéressant de recompiler ce sous-ensemble ou, au moins, d'ôter les classes inutilisées du JAR. La première approche est plus sûre dans la mesure où le compilateur vous garantit que vous ne supprimerez pas une partie essentielle du code, mais elle peut être assez délicate.

Ant et JAR

Vous avez deux possibilités pour intégrer du code tiers dans votre projet : utiliser du code source ou des JAR déjà assemblés.

Si vous choisissez la première méthode, il suffit de copier le code source dans l'arborescence de votre projet (sous le répertoire `src/`) afin qu'il soit placé à côté de votre propre code, puis de laisser le compilateur faire son travail.

Si vous choisissez d'utiliser un JAR dont vous ne possédez peut-être pas les sources, vous devrez expliquer à votre chaîne de développement comment l'utiliser. Commencez par placer le fichier JAR dans le répertoire `libs/` de votre projet Android. Puis, si vous vous servez d'un IDE, ajoutez le JAR à la liste des références nécessaires à votre projet. Si vous utilisez `ant` en ligne de commande, celui-ci utilisera automatiquement tous les fichiers JAR présents dans le répertoire `libs/`.

Suivre le script

À la différence des autres systèmes pour terminaux mobiles, Android n'impose aucune restriction sur ce qui peut s'exécuter tant que c'est du Java qui utilise la machine virtuelle Dalvik. Vous pouvez donc incorporer votre propre langage de script dans votre application, ce qui est expressément interdit sur d'autres terminaux.

BeanShell[1] est l'un de ces langages de script Java. Il offre une syntaxe compatible Java, avec un typage implicite, et ne nécessite pas de compilation.

Pour ajouter BeanShell, vous devez placer le fichier JAR de l'interpréteur dans votre répertoire `libs/`. Malheureusement, le JAR 2.0b4 disponible au téléchargement sur le site de BeanShell ne fonctionne pas tel quel avec Android 0.9 et les SDK plus récents, probablement à cause du compilateur utilisé pour le compiler. Il est donc préférable de récupérer son code source à l'aide de Subversion[2], d'exécuter `ant jarcore` pour le compiler et de copier le JAR ainsi obtenu (dans le répertoire `dist/` de BeanShell) dans le répertoire `libs/` de votre projet. Vous pouvez également utiliser le JAR BeanShell accompagnant les codes sources de ce livre (il se trouve dans le projet `Java/AndShell`).

Ensuite, l'utilisation de BeanShell avec Android est identique à son utilisation dans un autre environnement Java :

1. On crée une instance de la classe `Interpreter` de BeanShell.
2. On configure les variables globales pour le script à l'aide d'`Interpreter#set()`.
3. On appelle `Interpreter#eval()` pour lancer le script et, éventuellement, obtenir le résultat de la dernière instruction.

Voici par exemple le fichier de description XML du plus petit IDE BeanShell du monde :

```
<?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/eval"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Go !"
    />
```

1. http://beanshell.org.
2. http://beanshell.org/developer.html.

```
<EditText
    android:id="@+id/script"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:singleLine="false"
    android:gravity="top"
    />
</LinearLayout>
```

Voici l'implémentation de l'activité :

```
package com.commonsware.android.andshell;

import android.app.Activity;
import android.app.AlertDialog;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;
import bsh.Interpreter;

public class MainActivity extends Activity {
  private Interpreter i=new Interpreter();

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

    btn.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        String src=script.getText().toString();

        try {
          i.set("context", MainActivity.this);
          i.eval(src);
        }
        catch (bsh.EvalError e) {
          AlertDialog.Builder builder=
                    new AlertDialog.Builder(MainActivity.this);
```

```
                builder
                  .setTitle("Exception !")
                  .setMessage(e.toString())
                  .setPositiveButton("OK", null)
                  .show();
              }
            }
          });
        }
      }
```

Compilez ce projet (en incorporant le JAR de BeanShell comme on l'a mentionné plus haut), puis installez-le sur l'émulateur. Lorsque vous le lancerez, vous obtiendrez un IDE très simple, avec une grande zone de texte vous permettant de saisir votre script et un gros bouton `Go !` pour l'exécuter (voir Figure 24.1).

Figure 24.1
L'IDE AndShell.

Le script que nous voulons exécuter dans cet IDE est le suivant :

```
import android.widget.Toast;
Toast.makeText(context, "Hello, world!", 5000).show();
```

Notez l'utilisation de `context` pour désigner l'activité lors de la création du toast. Cette variable a été configurée globalement par l'activité pour se désigner elle-même. Vous pourriez l'appeler autrement : ce qui importe est que l'appel à `set()` et le code du script utilisent le même nom.

Lorsque vous cliquez sur le bouton `Go !`, vous obtenez le résultat de la Figure 24.2.

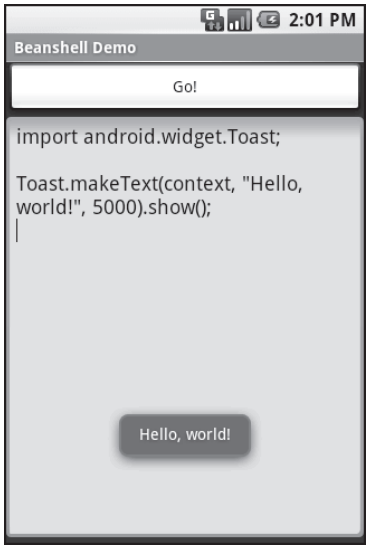


Figure 24.2
L'IDE AndShell exécutant un script BeanShell.

Ceci étant dit, il y a quelques limites à connaître :

- Tous les langages de script ne fonctionneront pas. Ceux qui implémentent leur propre forme de compilation JIT (*just-in-time*) – en produisant le pseudo-code Java à la volée –, notamment, devront sûrement être modifiés pour produire du pseudo-code Dalvik à la place. Les langages plus simples, qui interprètent seulement les fichiers de script en appelant les API d'introspection de Java pour se ramener à des appels de classes compilées, fonctionneront probablement mieux. Même en ce cas, certaines fonctionnalités du langage peuvent ne pas être disponibles si elles reposent sur une caractéristique de l'API Java traditionnelle qui n'existe pas avec Dalvik – ce qui peut être le cas du BeanShell ou de certains JAR tiers avec les versions actuelles d'Android.
- Les langages de script sans JIT seront forcément plus lents que des applications Dalvik compilées ; cette lenteur peut déplaire aux utilisateurs et impliquer plus de consommation de batterie pour le même travail. Construire une application Android en BeanShell uniquement parce que vous trouvez qu'elle est plus facile à écrire peut donc rendre vos utilisateurs assez mécontents.
- Les langages de script qui exposent toute l'API Java, comme BeanShell, peuvent réaliser tout ce qu'autorise le modèle de sécurité sous-jacent d'Android. Si votre application dispose de la permission `READ_CONTACTS`, par exemple, tous les scripts BeanShell qu'elle exécutera l'auront également (les permissions sont présentées au Chapitre 28).
- Enfin, mais ce n'est pas le moins important, les JAR des interpréteurs ont tendance à être... gros. Celui du BeanShell utilisé ici fait 200 Ko, par exemple. Ce n'est pas ridicule

si l'on considère ce qu'il est capable de faire, mais ceci implique que les applications qui utilisent BeanShell seront bien plus longues à télécharger, qu'elles prendront plus de place sur le terminal, etc.

Tout fonctionne... enfin, presque

Comme nous l'avons mentionné, tous les codes Java ne fonctionneront pas avec Android et Dalvik. Vous devez plus précisément tenir compte des paramètres suivants :

- Si le code Java suppose qu'il s'exécute avec Java SE, Java ME ou Java EE, il ne trouvera peut-être pas certaines API qu'il a l'habitude de trouver sur ces plates-formes mais qui ne sont pas disponibles avec Android. Certaines bibliothèques de tracé de diagrammes supposent, par exemple, la présence de primitives de tracé Swing ou AWT (*Abstract Window Toolkit*), qui sont généralement absentes d'Android.
- Le code Java peut dépendre d'un autre code Java qui, à son tour, peut avoir des problèmes pour s'exécuter sur Android. Vous pourriez vouloir utiliser un JAR qui repose, par exemple, sur une version de la classe `HTTPComponents` d'Apache plus ancienne (ou plus récente) que celle fournie avec Android.
- Le code Java peut utiliser des fonctionnalités du langage que le moteur Dalvik ne reconnaît pas.

Dans toutes ces situations, vous pouvez ne rencontrer aucun problème lors de la compilation de votre application avec un JAR compilé ; ces problèmes surviendront plutôt lors de l'exécution. C'est pour cette raison qu'il est préférable d'utiliser du code open-source avec Android à chaque fois que cela est possible : vous pourrez ainsi construire vous-même le code tiers en même temps que le vôtre et détecter plus tôt les difficultés à résoudre.

Autres langages de scripts

Ce chapitre étant consacré à l'écriture des scripts avec Android, vous apprécierez sûrement de savoir qu'il existe d'autres possibilités que l'intégration directe de Beanshell dans votre projet.

Certains essais ont été réalisés avec d'autres langages reposant sur la JVM, notamment JRuby et Jython. Pour le moment, leur support d'Android est incomplet, mais cette intégration progresse.

En outre, ASE (*Android Scripting Environment)*, téléchargeable à partir d'Android Market, permet d'écrire des scripts en Python et Lua, et de les faire exécuter par BeanShell. Ces scripts ne sont pas des applications à part entière et, à l'heure où ce livre est écrit, elles ne

sont pas vraiment redistribuables. De plus, ASE n'a pas été réellement conçu pour étendre d'autres applications, même s'il peut être utilisé de cette façon. Cependant, si vous voulez programmer directement sur le terminal, il est sûrement la meilleure solution actuelle. Pour plus d'informations sur ASE, consultez la page d'accueil du projet : **http://code.google.com/p/android-scripting/**.

25

Communiquer *via* Internet

On s'attend généralement à ce que la plupart des terminaux Android, si ce n'est tous, intègrent un accès à Internet. Cet accès peut passer par le Wifi, les services de données cellulaires (EDGE, 3G, etc.) ou, éventuellement, un mécanisme totalement différent. Quoi qu'il en soit, la plupart des gens – en tout cas, ceux qui ont un accès données ou Wifi – peuvent accéder à Internet à partir de leur téléphone Android.

Il n'est donc pas étonnant qu'Android offre aux développeurs un large éventail de moyens leur permettant d'exploiter cet accès. Ce dernier peut être de haut niveau, comme le navigateur `WebKit` intégré que nous avons étudié au Chapitre 13, mais il peut également intervenir au niveau le plus bas et utiliser des sockets brutes. Entre ces deux extrémités, il existe des API – disponibles sur le terminal ou *via* des JAR tiers – donnant accès à des protocoles spécifiques comme HTTP, XMPP, SMTP, etc.

Ce livre s'intéresse plutôt aux accès de haut niveau comme le composant `WebKit` et les API Internet car, dans la mesure du possible, les développeurs devraient s'efforcer de réutiliser des composants existants au lieu de créer leurs propres protocoles.

REST et relaxation

Bien qu'Android ne dispose pas d'API cliente pour SOAP ou XML-RPC, il intègre la bibliothèque `HttpComponents` d'Apache. Vous pouvez donc soit ajouter une couche SOAP/XML-RPC au-dessus de cette bibliothèque, soit l'utiliser directement pour accéder aux services web de type REST. Dans ce livre, nous considérerons les "services web REST" comme de simples requêtes HTTP classiques avec des réponses aux formats XML, JSON, etc.

Vous trouverez des didacticiels plus complets, des FAQ et des HOWTO sur le site web de `HttpComponents`[1] : nous ne présenterons ici que les bases en montrant comment consulter les informations météorologiques.

Opérations HTTP *via HttpComponents*

Le composant `HttpClient` de `HttpComponents` gère pour vous toutes les requêtes HTTP. La première étape pour l'utiliser consiste évidemment à créer un objet. `HttpClient` étant une interface, vous devrez donc instancier une implémentation de celle-ci, comme `DefaultHttpClient`.

Ces requêtes sont enveloppées dans des instances de `HttpRequest`, chaque commande HTTP étant gérée par une implémentation différente de cette interface (`HttpGet` pour les requêtes GET, par exemple). On crée donc une instance d'une implémentation de `HttpRequest`, on construit l'URL à récupérer ainsi que les autres données de configuration (les valeurs des formulaires si l'on effectue une commande POST *via* `HttpPost`, par exemple) puis l'on passe la méthode au client pour qu'il effectue la requête HTTP en appelant `execute()`.

Ce qui se passe ensuite peut être très simple ou très compliqué. On peut obtenir un objet `HttpResponse` enveloppant un code de réponse (200 pour OK, par exemple), des en-têtes HTTP, etc. Mais on peut également utiliser une variante d'`execute()` qui prend en paramètre un objet `ResponseHandler<String>` : cet appel renverra simplement une représentation de la réponse sous forme de `String`. En pratique, cette approche est déconseillée car il est préférable de vérifier les codes de réponses HTTP pour détecter les erreurs ; cependant, pour les applications triviales comme les exemples de ce livre, la technique `ResponseHandler<String>` convient parfaitement.

Le projet `Internet/Weather`, par exemple, implémente une activité qui récupère les données météorologiques de votre emplacement actuel à partir du site Google Weather. Ces données sont converties en HTML puis passées à un widget `WebKit` qui se charge de les afficher. Nous laissons en exercice au lecteur la réécriture de ce programme pour qu'il utilise un `ListView`. En outre, cet exemple étant relativement long, nous ne présenterons ici

1. **http://hc.apache.org/**.

que les extraits de code en rapport avec ce chapitre ; les sources complets sont disponibles dans les exemples fournis avec ce livre[1].

Pour rendre tout ceci un peu plus intéressant, nous utilisons les services de localisation d'Android pour déterminer notre emplacement actuel. Les détails de fonctionnement de ce service sont décrits au Chapitre 32.

Lorsqu'un emplacement a été trouvé – soit au lancement, soit parce que nous avons bougé –, nous récupérons les données de Google Weather *via* la méthode `updateForecast()` :

```
private void updateForecast(Location loc) {
    String url = String.format(format, ""
        + (int) (loc.getLatitude() * 1000000), ""
        + (int) (loc.getLongitude() * 1000000));
    HttpGet getMethod = new HttpGet(url);

    try {
      ResponseHandler<String> responseHandler = new BasicResponseHandler();
      String responseBody = client.execute(getMethod, responseHandler);

      buildForecasts(responseBody);

      String page = generatePage();

      browser.loadDataWithBaseURL(null, page, "text/html", "UTF-8", null);
    } catch (Throwable t) {
      Toast.makeText(this, "La requete a echoue: " + t.toString(), 4000)
           .show();
    }
  }
```

La méthode `updateForecast()` prend un objet `Location` en paramètre, obtenu *via* le processus de mise à jour de la localisation. Pour l'instant, il suffit de savoir que `Location` dispose des méthodes `getLatitude()` et `getLongitude()`, qui renvoient, respectivement, la latitude et la longitude.

L'URL de Google Weather est stockée dans une ressource chaîne à laquelle nous ajoutons en cours d'exécution la latitude et la longitude. Nous construisons un objet `HttpGet` avec cette URL (l'objet `HttpClient` a été créé dans `onCreate()`) puis nous exécutons cette méthode. À partir de la réponse XML, nous construisons la page HTML des prévisions que nous transmettons au widget `WebKit`. Si l'objet `HttpClient` échoue avec une exception, nous indiquons l'erreur à l'aide d'un toast.

1. Reportez-vous à la page dédiée à cet ouvrage sur le site **www.pearson.fr**.

Traitement des réponses

La réponse que l'on obtient est dans un certain format – HTML, XML, JSON, etc. – et c'est à nous, bien sûr, de choisir l'information qui nous intéresse pour en tirer quelque chose d'utile. Dans le cas de `WeatherDemo`, nous voulons extraire l'heure de la prévision, la température et l'icône (qui représente les conditions météorologiques) afin de nous en servir pour produire une page HTML.

Android fournit trois analyseurs XML : l'analyseur DOM classique du W3C (`org.w3c.dom`), un analyseur SAX (`org.xml.sax`) et l'analyseur pull présenté au Chapitre 20 ; il dispose également d'un analyseur JSON (`org.json`).

Lorsque cela est possible, vous pouvez bien sûr utiliser du code Java tiers pour prendre en charge d'autres formats – un analyseur RSS/Atom, par exemple. L'utilisation du code tiers a été décrite au Chapitre 24.

Pour `WeatherDemo`, nous utilisons l'analyseur DOM du W3C dans notre méthode `buildForecasts()` :

```
void buildForecasts(String raw) throws Exception {
    DocumentBuilder builder = DocumentBuilderFactory.newInstance()
        .newDocumentBuilder();
    Document doc = builder.parse(new InputSource(new StringReader(raw)));
    NodeList forecastList = doc.getElementsByTagName("forecast_conditions");
    for (int i = 0; i < forecastList.getLength(); i++) {
      Element currentFore = (Element) forecastList.item(i);

      // Retrouvons le jour de la semaine
      String day = currentFore.getElementsByTagName("day_of_week")
          .item(0).getAttributes().item(0).getNodeValue();
      String lowTemp = currentFore.getElementsByTagName("low").item(0)
          .getAttributes().item(0).getNodeValue();
      String highTemp = currentFore.getElementsByTagName("high").item(0)
          .getAttributes().item(0).getNodeValue();
      String icon = currentFore.getElementsByTagName("icon").item(0)
          .getAttributes().item(0).getNodeValue();
      Forecast f = new Forecast();
      f.setDay(day);
      f.setLowTemp(lowTemp);
      f.setHighTemp(highTemp);
      f.setIcon(icon);
      forecasts.add(f);
    }
  }
```

Le code HTML est lu comme un `InputStream` et est fourni à l'analyseur DOM. Puis on recherche les éléments `forecast_conditions` et l'on remplit un ensemble de modèles

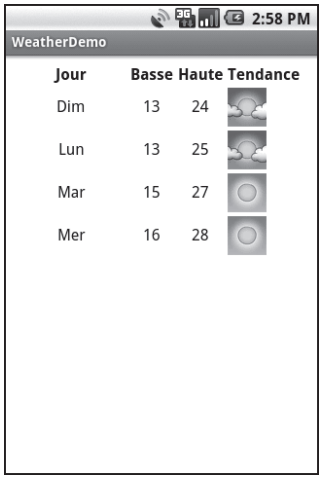
`Forecast` en utilisant la date, la température et l'URL de l'icône qui sera affichée en fonction du temps.

La méthode `generatePage()` produit à son tour un tableau HTML rudimentaire contenant les prévisions :

```
String generatePage() {
    StringBuffer bufResult = new StringBuffer("<html><body><table>");

    bufResult.append("<tr><th width=\"50%\">Jour</th>"
        + "<th>Basse</th><th>Haute</th><th>Tendance</th></tr>");

    for (Forecast forecast : forecasts) {
      bufResult.append("<tr><td align=\"center\">");
      bufResult.append(forecast.getDay());
      bufResult.append("</td><td align=\"center\">");
      bufResult.append(forecast.getLowTemp());
      bufResult.append("</td>");
      bufResult.append("</td><td align=\"center\">");
      bufResult.append(forecast.getHighTemp());
      bufResult.append("</td><td><img src=\"");
      bufResult.append(forecast.getIcon());
      bufResult.append("\"></td></tr>");
    }

    bufResult.append("</table></body></html>");

    return (bufResult.toString());
  }
```

La Figure 25.1 montre le résultat obtenu.

Figure 25.1
L'application `WeatherDemo`.



Autres points importants

Si vous devez utiliser SSL, n'oubliez pas que la configuration de `HttpClient` n'inclut pas SSL par défaut car c'est à vous de décider comment gérer la présentation des certificats SSL – les acceptez-vous tous aveuglément, même ceux qui sont autosignés ou qui ont expiré ? Préférez-vous demander confirmation à l'utilisateur avant d'accepter les certificats un peu étranges ?

De même, `HttpClient` est conçu par défaut pour être utilisé dans une application mono-thread, bien qu'il puisse aisément être configuré pour travailler dans un contexte multithreads.

Pour tous ces types de problèmes, la meilleure solution consiste à consulter la documentation et le support disponibles sur le site web de `HttpClient`.

26

Utilisation d'un fournisseur de contenu *(content provider)*

Avec Android, toute `Uri` de schéma `content://` représente une ressource servie par un fournisseur de contenu. Les fournisseurs de contenu encapsulent les données en utilisant des instances d'`Uri` comme descripteurs – on ne sait jamais d'où viennent les données représentées par l'`Uri` et l'on n'a pas besoin de le savoir : la seule chose qui compte est qu'elles soient disponibles lorsqu'on en a besoin. Ces données pourraient être stockées dans une base de données SQLite ou dans des fichiers plats, voire récupérées à partir d'un terminal ou stockées sur un serveur situé très loin d'ici, sur Internet.

À partir d'une `Uri`, vous pouvez réaliser les opérations CRUD de base (*Create, Read, Update, Delete*) en utilisant un fournisseur de contenu. Les instances d'`Uri` peuvent représenter des collections ou des éléments individuels. Grâce aux premières, vous pouvez créer

de nouveaux contenus *via* des opérations d'insertion. Avec les secondes, vous pouvez lire les données qu'elles représentent, les modifier ou les supprimer.

Android permet d'utiliser des fournisseurs de contenu existants ou de créer les vôtres. Ce chapitre est consacré à leur utilisation ; le Chapitre 27 expliquera comment mettre à disposition vos propres données à l'aide du framework des fournisseurs de contenu.

Composantes d'une *Uri*

Le modèle simplifié de construction d'une `Uri` est constitué du schéma, de l'espace de noms des données et, éventuellement, de l'identifiant de l'instance. Ces différents composants sont séparés par des barres de fraction, comme dans une URL. Le schéma d'une `Uri` de contenu est toujours `content://`.

L'`Uri` `content://constants/5` représente donc l'instance `constants` d'identifiant 5.

La combinaison du schéma et de l'espace de noms est appelée "Uri de base" d'un fournisseur de contenu ou d'un ensemble de données supporté par un fournisseur de contenu. Dans l'exemple précédent, `content://constants` est l'`Uri` de base d'un fournisseur de contenu qui sert des informations sur "constants" (en l'occurrence, des constantes physiques).

L'`Uri` de base peut être plus compliquée. Celle des contacts est, par exemple, `content://com.android.contacts/contacts`, car le fournisseur de contenu des contacts peut fournir d'autres données en utilisant d'autres valeurs pour l'`Uri` de base.

L'`Uri` de base représente une collection d'instances. Combinée avec un identifiant d'instance (5, par exemple), elle représente une instance unique.

La plupart des API d'Android s'attendent à ce que les URI soient des objets `Uri`, bien qu'il soit plus naturel de les considérer comme des chaînes. La méthode statique `Uri.parse()` permet de créer une instance d'`Uri` à partir de sa représentation textuelle.

Obtention d'un descripteur

D'où viennent ces instances d'`Uri` ?

Le point de départ le plus courant, lorsque l'on connaît le type de données avec lequel on souhaite travailler, consiste à obtenir l'`Uri` de base du fournisseur de contenu lui-même. `CONTENT_URI`, par exemple, est l'`Uri` de base des contacts représentés par des personnes – elle correspond à `content://com.android.contacts/contacts`. Si vous avez simplement besoin de la collection, cette `Uri` fonctionne telle quelle ; si vous avez besoin d'une instance dont vous connaissez l'identifiant, vous pouvez l'ajouter à la fin de cette dernière, afin d'obtenir une `Uri` pour cette instance précise.

Vous pourriez également obtenir des instances d'`Uri` à partir d'autres sources – vous pouvez récupérer des descripteurs d'`Uri` pour les contacts *via* des sous-activités répondant aux intentions `ACTION_PICK`, par exemple. Dans ce cas, l'`Uri` est vraiment un descripteur opaque... jusqu'à ce que vous décidiez de la déchiffrer à l'aide des différentes méthodes d'accès de la classe `Uri`.

Vous pouvez également utiliser des objets `String` codés en dur (`"content:// com.android.contacts/contacts"`, par exemple) et les convertir en instances `Uri` grâce à `Uri.parse()`. Ceci dit, ce n'est pas une solution idéale car les valeurs des `Uri` de base sont susceptibles d'évoluer au cours du temps.

Création des requêtes

À partir d'une `Uri` de base, vous pouvez exécuter une requête pour obtenir des données du fournisseur de contenu lié à cette `Uri`. Ce mécanisme ressemble beaucoup à SQL : on précise les "colonnes" qui nous intéressent, les contraintes permettant de déterminer les lignes du résultat, un ordre de tri, etc. La seule différence est que cette requête s'adresse à un fournisseur de contenu, pas directement à un SGBDR comme SQLite.

Le point névralgique de ce traitement est la méthode `managedQuery()`, qui attend cinq paramètres :

- L'`Uri` de base du fournisseur de contenu auquel s'adresse la requête ou l'`Uri` d'instance de l'objet interrogé.
- Un tableau des propriétés d'instances que vous voulez obtenir de ce fournisseur de contenu.
- Une contrainte, qui fonctionne comme la clause `WHERE` de SQL.
- Un ensemble éventuel de paramètres à lier à la contrainte, par remplacement des éventuels marqueurs d'emplacements qu'elle contient.
- Une instruction de tri facultative, qui fonctionne comme la clause `ORDER BY` de SQL.

Cette méthode renvoie un objet `Cursor` à partir duquel vous pourrez ensuite récupérer les données produites par la requête.

Les "propriétés" sont aux fournisseurs de contenu ce que sont les colonnes aux bases de données. En d'autres termes, chaque instance (ligne) renvoyée par une requête est formée d'un ensemble de propriétés (colonnes) représentant, chacune, un élément des données.

Tout ceci deviendra plus clair avec un exemple : voici l'appel de la méthode `managedQuery()` de la classe `ConstantsBrowser` du projet `ContentProvider/ConstantsPlus` :

```
constantsCursor=managedQuery(Provider.Constants.CONTENT_URI,
                             PROJECTION, null, null, null);
```

Les paramètres de cet appel sont :

- l'`Uri` passée à l'activité par l'appelant (`CONTENT_URI`), qui représente ici l'ensemble des constantes physiques gérées par le fournisseur de contenu ;
- la liste des propriétés à récupérer (voir le code ci-après) ;
- trois valeurs `null`, indiquant que nous n'utilisons pas de contrainte (car l'`Uri` représente l'instance que nous voulons), ni de paramètres de contrainte, ni de tri (nous ne devrions obtenir qu'une seule ligne).

```
private static final String[] PROJECTION = new String[] {
    Provider.Constants._ID, Provider.Constants.TITLE,
    Provider.Constants.VALUE};
```

Le plus gros "tour de magie", ici, est la liste des propriétés. Cette liste devrait être précisée dans la documentation (ou le code source) de chaque fournisseur de contenu. Ici, nous utilisons des valeurs logiques de la classe `Provider` qui représentent les différentes propriétés qui nous intéressent (l'identifiant unique, le nom et la valeur de la constante).

S'adapter aux circonstances

Lorsque `managedQuery()` nous a fourni un `Cursor`, nous avons accès au résultat de la requête et pouvons en faire ce que nous voulons. Nous pouvons, par exemple, extraire manuellement les données du `Cursor` pour remplir des widgets ou d'autres objets.

Cependant, si le but de la requête est d'obtenir une liste dans laquelle l'utilisateur pourra choisir un élément, il est préférable d'utiliser la classe `SimpleCursorAdapter`, qui établit un pont entre un `Cursor` et un widget de sélection comme une `ListView` ou un `Spinner`. Pour ce faire, il suffit de copier le `Cursor` dans un `SimpleCursorAdapter` que l'on passe ensuite au widget – ce widget montrera alors les options disponibles.

La méthode `onCreate()` de `ConstantsBrowser`, par exemple, présente à l'utilisateur la liste des constantes physiques :

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  constantsCursor=managedQuery(Provider.Constants.CONTENT_URI,
                               PROJECTION, null, null, null);
  ListAdapter adapter=new SimpleCursorAdapter(this,
                        R.layout.row, constantsCursor,
                        new String[] {Provider.Constants.TITLE,
                                      Provider.Constants.VALUE},
```

```
                                new int[] {R.id.title, R.id.value});
        setListAdapter(adapter);
        registerForContextMenu(getListView());
    }
```

Après l'exécution de `managedQuery()` et l'obtention du `Cursor`, `ConstantsBrowser` crée un `SimpleCursorAdapter` avec les paramètres suivants :

- L'activité (ou un autre `Context`) qui crée l'adaptateur. Ici, il s'agit de `ConstantsBrowser` elle-même.
- L'identifiant du layout utilisé pour afficher les éléments de la liste (`R.layout.row`).
- Le curseur (`constantsCursor`).
- Les propriétés à extraire du curseur et à utiliser pour configurer les instances `View` des éléments de la liste (`TITLE` et `VALUE`).
- Les identifiants correspondants des widgets `TextView` qui recevront ces propriétés (`R.id.title` et `R.id.value`).

Puis on place l'adaptateur dans la `ListView` et l'on obtient le résultat présenté à la Figure 26.1.

Pour disposer de plus de contrôle sur les vues, vous pouvez créer une sous-classe de `SimpleCursorAdapter` et redéfinir `getView()` afin de créer vos propres widgets pour la liste, comme on l'a expliqué au Chapitre 8.

Vous pouvez également manipuler manuellement le `Cursor` (avec `moveToFirst()` et `getString()`, par exemple) comme on l'a montré au Chapitre 22.

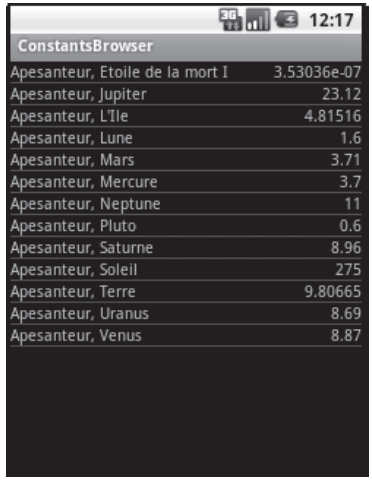


Figure 26.1
`ConstantsBrowser`, *affichage d'une liste de constantes physiques.*

Insertions et suppressions

Les fournisseurs de contenu seraient assez peu intéressants si vous ne pouviez pas ajouter ou supprimer des données et que vous deviez vous contenter de modifier celles qui s'y trouvent.

Pour insérer des données dans un fournisseur de contenu, l'interface `ContentProvider` (que vous pouvez obtenir *via* un appel à `getContentProvider()` dans votre activité) offre deux possibilités :

- La méthode `insert()` prend en paramètre une `Uri` de collection et une structure `ContentValues` décrivant l'ensemble de données à placer dans la ligne.
- La méthode `bulkInsert()` prend en paramètre une `Uri` de collection et un tableau de structures `ContentValues` pour remplir plusieurs lignes à la fois.
- La méthode `insert()` renvoie une `Uri` que vous pourrez manipuler ensuite. La méthode `bulkInsert()` renvoie le nombre de lignes créées ; vous devrez effectuer une requête pour obtenir à nouveau les données que vous venez d'insérer.

Voici, par exemple, un extrait de `ConstantsBrowser` qui insère une nouvelle constante au fournisseur de contenu à partir d'un `DialogWrapper` fournissant le nom et la valeur de cette constante :

```
private void processAdd(DialogWrapper wrapper) {
  ContentValues values=new ContentValues(2);
  values.put(Provider.Constants.TITLE, wrapper.getTitle());
  values.put(Provider.Constants.VALUE, wrapper.getValue());
  getContentResolver().insert(Provider.Constants.CONTENT_URI,
                              values);
  constantsCursor.requery();
}
```

Comme nous avons déjà un `Cursor` en suspens pour le contenu de ce fournisseur, nous appelons `requery()` sur celui-ci pour mettre à jour son contenu. Cet appel, à son tour, mettra à jour le `SimpleCursorAdapter` qui enveloppe éventuellement le `Cursor` – et ces modifications seront répercutées sur les widgets de sélection (`ListView`, par exemple) qui utilisent cet adaptateur.

Pour supprimer une ou plusieurs lignes d'un fournisseur de contenu, utilisez la méthode `delete()` de `ContentResolver`. Elle fonctionne comme l'instruction `DELETE` de SQL et prend trois paramètres :

- L'`Uri` représentant la collection (ou l'instance) que vous voulez supprimer.

- Une contrainte fonctionnant comme une clause `WHERE`, qui sert à déterminer les lignes qui doivent être supprimées.
- Un éventuel ensemble de paramètres qui remplaceront les marqueurs d'emplacements apparaissant dans la contrainte.

Attention aux BLOB !

Les BLOB (*Binary large objects*) existent dans de nombreux SGBDR, dont SQLite. Le modèle Android, cependant, préfère gérer ces volumes de données *via* leurs propres `Uri`. Un fournisseur de contenu, par conséquent, ne fournit pas d'accès direct *via* un `Cursor` aux données binaires comme les photos : à la place, c'est une propriété de ce fournisseur qui vous donnera l'`Uri` de ce BLOB particulier. Pour lire et écrire les données binaires, utilisez les méthodes `getInputStream()` et `getOutputStream()` de votre fournisseur de contenu.

Dans la mesure du possible, il vaut mieux minimiser les copies de données inutiles. L'utilisation principale d'une photo dans Android, par exemple, consiste à l'afficher, ce que sait parfaitement faire le widget `ImageView` si on lui fournit une `Uri` vers un fichier JPEG. En stockant la photo de sorte qu'elle possède sa propre `Uri`, vous n'avez pas besoin de copier des données du fournisseur de contenu vers une zone temporaire juste pour pouvoir l'afficher : il suffit d'utiliser l'`Uri`. On suppose, en fait, que peu d'applications Android feront beaucoup plus que déposer des données binaires et utiliser des widgets ou des activités prédéfinies pour afficher ces données.

27

Construction d'un fournisseur de contenu

La construction d'un fournisseur de contenu est sûrement la partie la plus compliquée et la plus ennuyeuse du développement avec Android car un fournisseur de contenu a de nombreuses exigences en termes d'implémentation de méthodes et d'exposition de données publiques. Malheureusement, tant que vous n'aurez pas utilisé votre fournisseur, vous ne pourrez pas savoir si vous avez tout fait correctement (c'est donc différent de la construction d'une activité, où le compilateur vous indique les erreurs que vous avez commises).

Ceci étant dit, la création d'un fournisseur de contenu est d'une importance capitale lorsqu'une application souhaite mettre ses données à disposition d'autres applications. Si elle ne les garde que pour elle-même, vous pouvez éviter la création d'un fournisseur de contenu en vous contentant d'accéder directement aux données depuis vos activités. En revanche, si vous souhaitez que vos données puissent être utilisées par d'autres – si,

par exemple, vous développez un lecteur RSS et que vous vouliez autoriser les autres programmes à accéder aux flux que vous avez téléchargés et mis en cache –, vous aurez besoin d'un fournisseur de contenu.

Pour illustrer notre propos, nous utiliserons ici l'application `ContentProvider/ConstantsPlus`, qui a déjà été présentée au Chapitre 26 et reprend le principe de l'application du Chapitre 22, mais en déplaçant le code qui manipule la base de données dans un fournisseur de contenu qui est ensuite utilisé par l'application.

D'abord, une petite dissection

Comme on l'a expliqué au chapitre précédent, l'`Uri` est le pilier de l'accès aux données d'un fournisseur de contenu : c'est la seule information qu'il faut réellement connaître. À partir de l'`Uri` de base du fournisseur, vous pouvez exécuter des requêtes ou construire une `Uri` vers une instance précise dont vous connaissez l'identifiant.

Cependant, pour construire un fournisseur de contenu, vous devez en savoir un peu plus sur les détails internes de l'`Uri` d'un contenu. Celle-ci est composée de deux à quatre parties en fonction de la situation :

- Elle comprend toujours un schéma (`content://`), indiquant qu'il s'agit d'une `Uri` de contenu, pas d'une `Uri` vers une ressource web (`http://`), par exemple.
- Elle a toujours une autorité, qui est la première partie du chemin placé après le schéma. L'autorité est une chaîne unique identifiant le fournisseur qui gère le contenu associé à cette `Uri`.
- Elle contient éventuellement un chemin de types de données, formé de la liste des segments de chemins situés entre l'autorité et l'identifiant d'instance (s'il y en a un). Ce chemin peut être vide si le fournisseur ne gère qu'un seul type de données. Il peut être formé d'un seul segment (`truc`) ou de plusieurs (`truc/machin/chouette`) pour gérer tous les scénarios d'accès aux données requis par le fournisseur de contenu.
- Elle peut contenir un identifiant d'instance, qui est un entier identifiant une information particulière du contenu. Une `Uri` de contenu sans identifiant d'instance désigne l'ensemble du contenu représenté par l'autorité (et, s'il est fourni, par le chemin des données).

Une `Uri` de contenu peut être aussi simple que `content://sekrits`, qui désigne l'ensemble du contenu fourni par n'importe quel fournisseur lié à l'autorité `sekrits` (`SecretsProvider`, par exemple), ou aussi compliquée que `content://sekrits/card/pin/17`, qui désigne l'information (identifiée par 17) de type `card/pin` gérée par le fournisseur de contenu `sekrits`.

Puis un peu de saisie

Vous devez ensuite proposer des types MIME correspondant au contenu de votre fournisseur.

Android utilise à la fois l'`Uri` de contenu et le type MIME pour identifier le contenu sur le terminal. Une `Uri` de collection – ou, plus précisément, la combinaison d'une autorité et d'un chemin de type de données – doit correspondre à deux types MIME, l'un pour la collection, l'autre pour une instance donnée. Ces deux types correspondent aux motifs d'`Uri` que nous avons vus dans la section précédente, respectivement pour les `Uri` avec et sans identifiant. Comme on l'a déjà vu, on peut fournir un type MIME à une intention pour qu'elle se dirige vers l'activité adéquate (l'intention `ACTION_PICK` sur un type MIME de collection pour appeler une activité de sélection permettant de choisir une instance dans cette collection, par exemple).

Le type MIME de la collection doit être de la forme `vnd.`*`X`*`.cursor.dir/`*`Y`*, où *X* est le nom de votre société, organisation ou projet et où *Y* est un nom de type délimité par des points. Vous pourriez, par exemple, utiliser le type MIME `vnd.tlagency.cursor.dir/sekrits.card.pin` pour votre collection de secrets.

Le type MIME de l'instance doit être de la forme `vnd.`*`X`*`.cursor.item/`*`Y` où *X* et *Y* sont généralement les mêmes valeurs que celles utilisées pour le type MIME de la collection (bien que ce ne soit pas obligatoire).

Création d'un fournisseur de contenu

La création d'un fournisseur de contenu s'effectue en quatre étapes : création de la classe du fournisseur, fourniture d'une `Uri`, déclaration des propriétés et modification du manifeste.

Étape n° 1 : création d'une classe *Provider*

Un fournisseur de contenu est une classe Java, tout comme une activité et une intention. La principale étape de la création d'un fournisseur consiste donc à produire sa classe Java, qui doit hériter de `ContentProvider`.

Cette sous-classe doit implémenter six méthodes qui, ensemble, assurent le service qu'un fournisseur de contenu est censé offrir aux activités qui veulent créer, lire, modifier ou supprimer du contenu.

onCreate()

Comme pour une activité, le point d'entrée principal d'un fournisseur de contenu est sa méthode `onCreate()`, où l'on réalise toutes les opérations d'initialisation que l'on souhaite.

C'est là, notamment, que l'on initialise le stockage des données. Si, par exemple, on compte stocker les données dans tel ou tel répertoire d'une carte SD et utiliser un fichier XML comme "table des matières", c'est dans `onCreate()` que l'on vérifiera que ce répertoire et ce fichier existent et, dans le cas contraire, qu'on les créera pour que le reste du fournisseur de contenu sache qu'ils sont disponibles.

Voici, par exemple, la méthode `onCreate()` de la classe `Provider` du projet `ContentProvider/ConstantsPlus` :

```
@Override
public boolean onCreate() {
  db=(new DatabaseHelper(getContext())).getWritableDatabase();

  return (db == null) ? false : true;
}
```

Toute la "magie" de ce code se trouve dans l'objet privé `DatabaseHelper`, qui a été décrit au Chapitre 22.

query()

Comme l'on pourrait s'y attendre, la méthode `query()` contient le code grâce auquel le fournisseur de contenu obtient des détails sur une requête qu'une activité veut réaliser. C'est à vous de décider d'effectuer ou non cette requête.

La méthode `query()` attend les paramètres suivants :

- une `Uri` représentant la collection ou l'instance demandée ;
- un tableau de `String` contenant la liste des propriétés à renvoyer ;
- un `String` représentant l'équivalent d'une clause `WHERE` de SQL et qui contraint le résultat de la requête ;
- un tableau de `String` contenant les valeurs qui remplaceront les éventuels marqueurs d'emplacements dans le paramètre précédent ;
- un `String` équivalant à une clause `ORDER BY` de SQL.

C'est vous qui êtes responsable de l'interprétation de ces paramètres et vous devez renvoyer un `Cursor` qui pourra ensuite être parcouru pour accéder aux données.

Comme vous pouvez vous en douter, ces paramètres sont fortement orientés vers l'utilisation de SQLite comme moyen de stockage. Vous pouvez en ignorer certains (la clause `WHERE`, par exemple) mais vous devez alors l'indiquer pour que les activités ne vous interrogent que par une `Uri` d'instance et n'utilisent pas les paramètres que vous ne gérez pas.

Pour les fournisseurs qui utilisent SQLite, toutefois, l'implémentation de la méthode `query()` devrait être triviale : il suffit d'utiliser un `SQLiteQueryBuilder` pour convertir les différents paramètres en une seule instruction SQL, puis d'appeler la méthode `query()` de cet objet pour invoquer la requête et obtenir le `Cursor` qui sera ensuite renvoyé par votre propre méthode `query()`.

Voici, par exemple, l'implémentation de la méthode `query()` de `Provider` :

```
@Override
public Cursor query(Uri url, String[] projection, String selection,
                    String[] selectionArgs, String sort) {
  SQLiteQueryBuilder qb=new SQLiteQueryBuilder();

  qb.setTables(getTableName());

  if (isCollectionUri(url)) {
    qb.setProjectionMap(getDefaultProjection());
  }
  else {
    qb.appendWhere(getIdColumnName() + "=" +
                   url.getPathSegments().get(1));
  }

  String orderBy;

  if (TextUtils.isEmpty(sort)) {
    orderBy=getDefaultSortOrder();
  } else {
    orderBy=sort;
  }

  Cursor c=qb.query(db, projection, selection, selectionArgs,
                    null, null, orderBy);
  c.setNotificationUri(getContext().getContentResolver(), url);
  return c;
}
```

Nous créons un `SQLiteQueryBuilder` et nous insérons les détails de la requête dans cet objet. Vous remarquerez que la requête pourrait utiliser soit une `Uri` de collection, soit une `Uri` d'instance – dans ce dernier cas, nous devons ajouter l'identifiant d'instance à la requête. Puis nous utilisons la méthode `query()` de l'objet builder pour obtenir un `Cursor` correspondant au résultat.

insert()

Votre méthode `insert()` recevra une `Uri` représentant la collection et une structure `ContentValues` contenant les données initiales de la nouvelle instance. C'est à vous de la créer, d'y placer les données fournies et de renvoyer une `Uri` vers cette nouvelle instance.

Une fois encore, cette implémentation sera triviale pour un fournisseur de contenu qui utilise SQLite : il suffit de vérifier que toutes les données requises ont été fournies par l'activité, de fusionner les valeurs par défaut avec les données fournies et d'appeler la méthode `insert()` de la base de données pour créer l'instance.

Voici, par exemple, l'implémentation de la méthode `insert()` de `Provider` :

```
@Override
public Uri insert(Uri url, ContentValues initialValues) {
  long rowID;
  ContentValues values;

  if (initialValues!=null) {
    values=new ContentValues(initialValues);
  } else {
    values=new ContentValues();
  }

  if (!isCollectionUri(url)) {
    throw new IllegalArgumentException("URL inconnue" + url);
  }

  for (String colName : getRequiredColumns()) {
    if (values.containsKey(colName) == false) {
      throw new IllegalArgumentException("Colonne manquante : " +
                                          colName);
    }
  }

  populateDefaultValues(values);

  rowID=db.insert(getTableName(), getNullColumnHack(), values);
  if (rowID > 0) {
    Uri uri=ContentUris.withAppendedId(getContentUri(), rowID);
    getContext().getContentResolver().notifyChange(uri, null);
    return uri;
  }
  throw new SQLException("Echec de l'insertion dans " + url);
}
```

La technique est la même que précédemment : pour réaliser l'insertion, on utilise les particularités du fournisseur, plus les données à insérer. Les points suivants méritent d'être notés :

- Une insertion ne peut se faire que dans une `Uri` de collection, c'est la raison pour laquelle on teste le paramètre avec `isCollectionUri()`.
- Le fournisseur sachant quelles sont les colonnes requises (avec `getRequiredColumns()`), nous les parcourons et nous vérifions que les valeurs fournies leur correspondent.

C'est au fournisseur de fournir les valeurs par défaut (*via* `populateDefaultValues()`) pour les colonnes qui ne sont pas fournies à l'appel d'`insert()` et qui ne sont pas automatiquement gérées par la définition de la table SQLite.

update()

La méthode `update()` prend en paramètre l'`Uri` de l'instance ou de la collection à modifier, une structure `ContentValues` contenant les nouvelles valeurs, une chaîne correspondant à une clause `WHERE` de SQL et un tableau de chaînes contenant les valeurs qui remplaceront les marqueurs d'emplacement dans la clause `WHERE`. Il vous appartient d'identifier l'instance ou les instances à modifier (en utilisant l'`Uri` et la clause `WHERE`) puis de remplacer leurs valeurs actuelles par celles qui ont été fournies en paramètre.

Si vous utilisez SQLite comme système de stockage, il suffit de transmettre tous les paramètres à la méthode `update()` de la base, même si cet appel variera légèrement selon que vous modifiez une ou plusieurs instances.

Voici, par exemple, l'implémentation de la méthode `update()` de `Provider` :

```
@Override
public int update(Uri url, ContentValues values, String where,
                  String[] whereArgs) {
  int count;

  if (isCollectionUri(url)) {
    count=db.update(getTableName(), values, where, whereArgs);
  }
  else {
    String segment=url.getPathSegments().get(1);
    count=db
        .update(getTableName(), values, getIdColumnName() + "="
            + segment
            + (!TextUtils.isEmpty(where) ? " AND (" + where
```

```
                + ')' : ""), whereArgs);
    }

    getContext().getContentResolver().notifyChange(url, null);
    return count;
  }
```

Ici, les modifications pouvant s'appliquer à une instance précise ou à toute la collection, nous testons l'`Uri` avec `isCollectionUri()` : s'il s'agit d'une modification de collection, nous nous contentons de la faire. S'il s'agit d'une modification d'une seule instance, nous ajoutons une contrainte à la clause `WHERE` pour que la requête ne porte que sur la ligne concernée.

delete()

Comme `update()`, `delete()` reçoit une `Uri` représentant l'instance ou la collection concernée, une clause `WHERE` et ses paramètres. Si l'activité supprime une seule instance, l'`Uri` doit la représenter et la clause `WHERE` peut être `null`. Mais l'activité peut également demander la suppression d'un ensemble d'instances en utilisant la clause `WHERE` pour indiquer les instances à détruire.

Comme pour `update()`, l'implémentation de `delete()` est simple si l'on utilise SQLite car nous pouvons laisser à ce dernier le soin d'analyser et d'appliquer la clause `WHERE` – la seule chose à faire est d'appeler la méthode `delete()` de la base de données.

Voici, par exemple, l'implémentation de la méthode `delete()` de `Provider` :

```
@Override
public int delete(Uri url, String where, String[] whereArgs) {
  int count;
  long rowId=0;

  if (isCollectionUri(url)) {
    count=db.delete(getTableName(), where, whereArgs);
  }
  else {
    String segment=url.getPathSegments().get(1);
    rowId=Long.parseLong(segment);
    count=db
        .delete(getTableName(), getIdColumnName() + "="
            + segment
            + (!TextUtils.isEmpty(where) ? " AND (" + where
                + ')' : ""), whereArgs);
```

```
    }

    getContext().getContentResolver().notifyChange(url, null);
    return count;
  }
```

Ce code est quasiment identique à celui de la méthode `update()` que nous avons décrite précédemment – il supprime un sous-ensemble de la collection ou une instance unique (si elle vérifie la clause `WHERE` passée en paramètre).

getType()

`getType()` est la dernière méthode qu'il faut implémenter. Elle prend une `Uri` en paramètre et renvoie le type MIME qui lui est associé. Cette `Uri` pouvant désigner une collection ou une instance, vous devez la tester pour renvoyer le type MIME correspondant.

Voici l'implémentation de la méthode `getType()` de `Provider` :

```
@Override
public String getType(Uri url) {
  if (isCollectionUri(url)) {
    return(getCollectionType());
  }

  return(getSingleType());
}
```

Comme vous pouvez le constater, l'essentiel de ce code est délégué aux méthodes privées `getCollectionType()` et `getSingleType()` :

```
private String getCollectionType() {
  return("vnd.android.cursor.dir/vnd.commonsware.constant");
}

private String getSingleType() {
  return("vnd.android.cursor.item/vnd.commonsware.constant");
}
```

Étape n° 2 : fournir une *Uri*

Vous devez également ajouter un membre public statique contenant l'`Uri` de chaque collection supportée par votre fournisseur de contenu. Généralement, on utilise une `Uri` constante, publique et statique dans la classe du fournisseur lui-même :

```
public static final Uri CONTENT_URI =
  Uri.parse("content://com.commonsware.android.constants.Provider/constants");
```

Vous pouvez utiliser le même espace de noms pour l'`Uri` de contenu que pour vos classes Java, afin de réduire le risque de collisions de noms.

Étape n° 3 : déclarer les propriétés

Pour définir les noms des propriétés de votre fournisseur, vous devez créer une classe publique statique implémentant l'interface `BaseColumns` :

```
public static final class Constants implements BaseColumns {
  public static final Uri CONTENT_URI =
    Uri.parse("content://com.commonsware.android.constants.Provider/constants");
  public static final String DEFAULT_SORT_ORDER="title";
  public static final String TITLE="title";
  public static final String VALUE="value";
}
```

Si vous utilisez SQLite pour stocker les données, les valeurs des constantes correspondant aux noms des propriétés doivent être les noms des colonnes respectives de la table, afin de pouvoir simplement passer la projection (tableau de propriétés) lors de l'appel à `query()` ou le `ContentValues` lors d'un appel à `insert()` ou `update()`.

Vous remarquerez que les types des propriétés ne sont pas précisés. Il pourrait s'agir de chaînes, d'entiers, etc. – en réalité, ils sont limités par les types autorisés par les méthodes d'accès du `Cursor`. Le fait que rien ne permette de tester les types signifie que vous devez documenter soigneusement vos propriétés afin que les utilisateurs de votre fournisseur de contenu sachent à quoi s'attendre.

Étape n° 4 : modifier le manifeste

Ce qui lie l'implémentation du fournisseur de contenu au reste de l'application se trouve dans le fichier `AndroidManifest.xml`. Il suffit d'ajouter un élément `<provider>` comme fils de l'élément `<application>` :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.constants">
  <application android:label="@string/app_name"
    android:icon="@drawable/cw">
    <provider android:name=".Provider"
        android:authorities="com.commonsware.android.constants.Provider"
    />
    <activity android:name=".ConstantsBrowser"
              android:label="@string/app_name">
      <intent-filter>
```

```
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
          </intent-filter>
        </activity>
      </application>
    </manifest>
```

La propriété `android:name` est le nom de la classe du fournisseur de contenu, préfixé par un point pour indiquer qu'il se trouve dans l'espace de noms des classes de cette application.

La propriété `android:authorities` est une liste des autorités reconnues par le fournisseur de contenu, séparées par des points-virgules. Plus haut dans ce chapitre, nous avons vu qu'une `Uri` de contenu était constituée d'un schéma, d'une autorité, d'un chemin de types de données et d'un identifiant d'instance. Chaque autorité de chaque valeur `CONTENT_URI` devrait être incluse dans la liste `android:authorities`.

Quand Android rencontrera une `Uri` de contenu, il pourra désormais passer en revue les fournisseurs enregistrés *via* les manifestes afin de trouver une autorité qui correspond. Il saura alors quelle application et quelle classe implémentent le fournisseur de contenu et pourra ainsi établir le lien entre l'activité appelante et le fournisseur de contenu appelé.

Avertissements en cas de modifications

Votre fournisseur de contenu peut éventuellement avertir ses clients lorsque des modifications ont été apportées aux données d'une `Uri` de contenu particulière.

Supposons, par exemple, que vous ayez créé un fournisseur de contenu qui récupère des flux RSS et Atom sur Internet en fonction des abonnements de l'utilisateur (*via* OPML, par exemple). Le fournisseur offre un accès en lecture seule au contenu des flux et garde un œil vers les différentes applications du téléphone qui les utilisent (ce qui est préférable à une solution où chacun implémenterait son propre système de récupération des flux). Vous avez également implémenté un service qui obtient de façon asynchrone les mises à jour de ces flux et qui modifie les données stockées sous-jacentes. Votre fournisseur de contenu pourrait alerter les applications qui utilisent les flux que tel ou tel flux a été mis à jour, afin que celles qui l'utilisent puissent rafraîchir ses données pour disposer de la dernière version.

Du côté du fournisseur de contenu, ceci nécessite d'appeler `notifyChange()` sur votre instance de `ContentResolver` (que vous pouvez obtenir *via* un appel à `getContext().getContentResolver()`). Cette méthode prend deux paramètres : l'`Uri` de la partie du contenu qui a été modifiée et le `ContentObserver` qui a initié la modification. Dans de nombreux cas, ce deuxième paramètre vaudra `null` ; une valeur non `null` signifie simplement que l'observateur qui a lancé la modification ne sera pas prévenu de sa propre modification.

Du côté du consommateur de contenu, une activité peut appeler `registerContentObserver()` sur son `ContentResolver` (obtenu *via* `getContentResolver()`). Cet appel lie une instance de `ContentObserver` à l'`Uri` fournie – l'observateur sera prévenu à chaque appel de `notifyChange()` pour cette `Uri`. Lorsque le consommateur en a fini avec l'`Uri`, un appel à `unregisterContentObserver()` permet de libérer la connexion.

28

Demander et exiger des permissions

À la fin des années 1990, une vague de virus s'est répandue sur Internet *via* les courriers électroniques. Ces virus utilisaient les informations stockées dans le carnet d'adresses de Microsoft Outlook : ils s'envoyaient simplement en copie à tous les contacts Outlook d'une machine infectée. Cette épidémie a été rendue possible parce que, à cette époque, Outlook ne prenait aucune mesure pour protéger ses données des programmes qui utilisaient son API car celle-ci avait été conçue pour des développeurs ordinaires, pas pour des auteurs de virus.

Aujourd'hui, de nombreuses applications qui utilisent des contacts les sécurisent en exigeant qu'un utilisateur donne le droit d'y accéder. Ces droits peuvent être octroyés au cas par cas ou une fois pour toutes, lors de l'installation.

Android fonctionne de la même façon : il exige que les applications qui souhaitent lire ou écrire les contacts aient les droits nécessaires. Cependant, son système de permission va bien au-delà des données de contact puisqu'il s'applique également aux fournisseurs de contenu et aux services qui ne font pas partie du framework initial.

En tant que développeur Android, vous devrez fréquemment vous assurer que vos applications aient les permissions adéquates pour manipuler les données des autres applications. Vous pouvez également choisir d'exiger des permissions pour que les autres applications aient le droit d'utiliser vos données ou vos services si vous les mettez à disposition des autres composants d'Android.

Maman, puis-je ?

Demander d'utiliser les données ou les services d'autres applications exige d'ajouter l'élément `uses-permission` au fichier `AndroidManifest.xml`. Votre manifeste peut ainsi contenir zéro ou plusieurs de ces éléments comme fils directs de l'élément racine `manifest`.

L'élément `uses-permission` n'a qu'un seul attribut, `android:name`, qui est le nom de la permission exigée par votre application :

```
<uses-permission
  android:name="android.permission.ACCESS_LOCATION" />
```

Les permissions système de base commencent toutes par `android.permission` et sont énumérées dans la documentation du SDK à la rubrique `Manifest.permission`. Les applications tierces peuvent posséder leurs propres permissions, qui seront sans doute également documentées. Voici quelques-unes des permissions prédéfinies les plus importantes :

- `INTERNET`, si votre application souhaite accéder à Internet par quelque moyen que ce soit, des sockets brutes de Java au widget `WebView`.
- `READ_CALENDAR`, `READ_CONTACTS`, etc. pour la lecture des données à partir des fournisseurs de contenu intégrés.
- `WRITE_CALENDAR`, `WRITE_CONTACTS`, etc. pour la modification des données dans les fournisseurs de contenu intégrés.

L'utilisateur devra confirmer ces permissions lors de l'installation de l'application. Il est important de ne demander que le minimum de permissions et de justifier cette demande afin que les utilisateurs ne choisissent pas d'ignorer votre application parce qu'elle exige trop de permissions inutiles. Notez toutefois que cette demande de confirmation n'est pas disponible dans la version actuelle de l'émulateur.

Si vous ne possédez pas une permission donnée et que vous tentez d'effectuer une opération qui l'exige, l'exception `SecurityException` vous informera du problème, bien que ce ne soit pas une garantie – cet échec peut prendre d'autres formes si cette exception est capturée et traitée par ailleurs. La vérification d'une permission n'échoue que si vous avez oublié de demander cette permission ; votre application ne peut pas s'exécuter sans que vous ayez obtenu les permissions qu'elle exige.

Halte ! Qui va là ?

L'autre côté de la médaille est, bien sûr, la sécurisation de votre propre application. Si celle-ci est uniquement constituée d'activités et de récepteurs d'intention, la sécurité peut être simplement tournée "vers l'extérieur", lorsque vous demandez le droit d'utiliser les ressources d'autres applications. Si, en revanche, votre application comprend des fournisseurs de contenu ou des services, vous voudrez mettre en place une sécurité tournée "vers l'intérieur", pour contrôler les applications qui peuvent accéder à vos données et le type de ces accès.

Le problème, ici, est moins les "perturbations" que peuvent causer les autres applications à vos données que la confidentialité des informations ou l'utilisation de services qui pourraient vous coûter cher. Les permissions de base d'Android sont conçues pour cela – pouvez-vous lire ou modifier des contacts, envoyer des SMS ? etc. Si votre application ne stocke pas des informations privées, la sécurité est moins un problème. Mais, si elle stocke des données personnelles, comme des informations médicales, la sécurité devient bien plus importante.

La première étape pour sécuriser votre propre application à l'aide de permissions consiste à les déclarer dans le fichier `AndroidManifest.xml`. Dans ce cas, au lieu d'utiliser `uses-permission`, vous ajoutez des éléments `permission`. Là encore, il peut y avoir zéro ou plusieurs de ces éléments, qui seront tous des fils directs de `manifest`.

Déclarer une permission est un peu plus compliqué qu'utiliser une permission car vous devez donner trois informations :

- **Le nom symbolique de la permission.** Pour éviter les collisions de vos permissions avec celles des autres applications, utilisez l'espace de noms Java de votre application comme préfixe.
- **Un label pour la permission :** un texte court et compréhensible par les utilisateurs.
- **Une description de la permission :** un texte un peu plus long et compréhensible par les utilisateurs.

```
<permission
  android:name="vnd.tlagency.sekrits.SEE_SEKRITS"
  android:label="@string/see_sekrits_label"
  android:description="@string/see_sekrits_description" />
```

Cet élément n'impose pas la permission. Il indique seulement qu'il s'agit d'une permission possible ; votre application doit quand même savoir détecter les violations de sécurité lorsqu'elles ont lieu.

Une application a deux moyens d'imposer des permissions, en précisant où et dans quelles circonstances elles sont requises. Le plus simple consiste à indiquer dans le manifeste les endroits où elles sont requises.

Imposer les permissions *via* le manifeste

Les activités, les services et les récepteurs d'intentions peuvent déclarer un attribut `android:permission` dont la valeur est le nom de la permission exigée pour accéder à ces éléments :

```
<activity
  android:name=".SekritApp"
  android:label="Top Sekrit"
  android:permission="vnd.tlagency.sekrits.SEE_SEKRITS">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category
      android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

Seules les applications qui ont demandé la permission indiquée pourront accéder au composant ainsi sécurisé. Ici, "accéder" à la signification suivante :

- Les activités ne peuvent pas être lancées sans cette permission.
- Les services ne peuvent pas être lancés, arrêtés ou liés à une activité sans cette permission.

Les récepteurs d'intention ignorent les messages envoyés *via* `sendBroadcast()` si l'expéditeur ne possède pas cette permission.

Les fournisseurs de contenu offrent deux attributs distincts, `android:readPermission` et `android:writePermission` :

```
<provider
  android:name=".SekritProvider"
  android:authorities="vnd.tla.sekrits.SekritProvider"
  android:readPermission="vnd.tla.sekrits.SEE_SEKRITS"
  android:writePermission="vnd.tla.sekrits.MOD_SEKRITS" />
```

Dans ce cas, `readPermission` contrôle l'accès à la consultation du fournisseur de contenu, tandis que `writePermission` contrôle l'accès aux insertions, aux modifications et aux suppressions de données dans le fournisseur.

Imposer les permissions ailleurs

Il y a deux moyens supplémentaires d'imposer les permissions dans le code Java.

- Vos services peuvent vérifier les permissions à chaque appel, grâce à `checkCallingPermission()`, qui renvoie `PERMISSION_GRANTED` ou `PERMISSION_DENIED` selon que l'appelant a ou non la permission indiquée. Si, par exemple, votre service implémente des méthodes de lecture et d'écriture séparées, vous pourriez obtenir le même effet que `readPermission` et `writePermission` en vérifiant que ces méthodes ont les permissions requises.
- Vous pouvez également inclure une permission dans l'appel à `sendBroadcast()`. Ceci signifie que les récepteurs possibles doivent posséder cette permission : sans elle, ils ne pourront pas recevoir l'intention. Le sous-système d'Android inclut probablement la permission `RECEIVE_SMS` lorsqu'il signale qu'un SMS est arrivé, par exemple – ceci restreint les récepteurs de cette intention à ceux qui sont autorisés à recevoir des messages SMS.

Papiers, s'il vous plaît !

Il n'y a pas de découverte automatique des permissions lors de la compilation ; tous les échecs liés aux permissions ont lieu à l'exécution. Il est donc important de documenter les permissions requises pour votre API publique, ce qui comprend les fournisseurs de contenu, les services et les activités conçues pour être lancées à partir d'autres activités. Dans le cas contraire, les programmeurs voulant s'interfacer avec votre application devront batailler pour trouver les règles de permissions que vous avez mises en place.

En outre, vous devez vous attendre à ce que les utilisateurs de votre application soient interrogés pour confirmer les permissions dont votre application a besoin. Vous devez donc leur indiquer à quoi s'attendre, sous peine qu'une question posée par le téléphone ne les décide à ne pas installer ou à ne pas utiliser l'application.

29

Création d'un service

Comme on l'a déjà mentionné, les services Android sont destinés aux processus de longue haleine, qui peuvent devoir continuer de s'exécuter même lorsqu'ils sont séparés de toute activité. À titre d'exemple, citons la musique, qui continue d'être jouée même lorsque l'activité de "lecture" a été supprimée, la récupération des mises à jour des flux RSS sur Internet et la persistance d'une session de messagerie instantanée, même lorsque le client a perdu le focus à cause de la réception d'un appel téléphonique.

Un service est créé lorsqu'il est lancé manuellement (*via* un appel à l'API) ou quand une activité tente de s'y connecter *via* une communication interprocessus (IPC). Il perdure jusqu'à ce qu'il ne soit plus nécessaire et que le système ait besoin de récupérer de la mémoire. Cette longue exécution ayant un coût, les services doivent prendre garde à ne pas consommer trop de CPU sous peine d'user trop rapidement la batterie du terminal.

Dans ce chapitre, vous apprendrez à créer vos propres services ; le chapitre suivant sera consacré à l'utilisation de ces services à partir de vos activités ou d'autres contextes. Tous les deux utiliseront comme exemple le projet `Service/WeatherPlus`, dont l'implémentation sera essentiellement présentée dans ce chapitre. Ce projet `WeatherPlusService` étend `Internet/Weather` en l'empaquetant dans un service qui surveille les modifications de l'emplacement du terminal, afin que les prévisions soient mises à jour lorsque celui-ci "se déplace".

Service avec classe

L'implémentation d'un service partage de nombreuses caractéristiques avec la construction d'une activité. On hérite d'une classe de base fournie par Android, on redéfinit certaines méthodes du cycle de vie et on accroche le service au système *via* le manifeste.

La première étape de création d'un service consiste à étendre la classe `Service` – en dérivant une classe `WeatherPlusService` dans le cadre de notre exemple.

Tout comme les activités peuvent redéfinir les méthodes `onCreate()`, `onResume()`, `onPause()` et apparentées, les implémentations de `Service` peuvent redéfinir trois méthodes du cycle de vie :

- `onCreate()`, qui, comme pour les activités, sera appelée lorsque le processus du service est créé.
- `onStart()`, qui est appelée à chaque fois qu'un service est lancé par `startService()`.
- `onDestroy()`, qui est appelée lorsque le service est éteint.

Voici, par exemple, la méthode `onCreate()` de `WeatherPlusService` :

```
@Override
public void onCreate() {
  super.onCreate();

  client=new DefaultHttpClient();
  format=getString(R.string.url);

  mgr =
(LocationManager)getSystemService(Context.LOCATION_SERVICE);
  mgr.requestLocationUpdates(LocationManager.GPS_PROVIDER,
                             10000,
10000.0f, onLocationChange);

}
```

On appelle d'abord la méthode `onCreate()` de la superclasse afin qu'Android puisse effectuer correctement son travail d'initialisation. Puis on crée notre `HttpClient` et la chaîne de format, comme nous l'avions fait dans le projet `Weather`. On obtient ensuite l'instance de `LocationManager` pour notre application et on lui demande de récupérer les mises à jour à mesure que notre emplacement évolue, *via* `LocationManager.GPS_MANAGER`, qui sera détaillé au Chapitre 32.

La méthode `onDestroy()` est bien plus simple :

```
@Override
public void onDestroy() {
```

```
        super.onDestroy();


        mgr.removeUpdates(onLocationChange);
    }
```

On se contente ici de stopper la surveillance des déplacements, après avoir appelé la méthode `onDestroy()` de la superclasse pour qu'Android puisse effectuer les travaux de nettoyage nécessaires.

Outre ces méthodes du cycle de vie, votre service doit également implémenter la méthode `onBind()`, qui renvoie un `IBinder`, la composante fondamentale du mécanisme d'IPC. Nous examinerons cette méthode plus en détail dans la section suivante.

Il ne peut en rester qu'un !

Par défaut, les services s'exécutent dans le même processus que tous les autres composants de l'application – les activités, par exemple. On peut donc appeler les méthodes de l'API sur l'objet service... à condition de mettre la main dessus. Dans l'idéal, il devrait exister un moyen de demander à Android de nous donner l'objet du service local ; malheureusement, ce moyen n'existe pas encore et nous en sommes donc réduits à tricher.

Il ne peut y avoir, au plus, qu'une seule copie d'un même service qui s'exécute en mémoire. Il peut n'y en avoir aucune si le service n'a pas été lancé mais, même si plusieurs activités tentent d'utiliser le service, un seul s'exécutera vraiment. Il s'agit donc d'une implémentation du patron de conception singleton – il suffit alors que l'on expose le singleton lui-même pour que les autres composants puissent accéder à l'objet.

Nous pourrions exposer ce singleton *via* un membre public statique ou une méthode getter publique et statique, mais nous prendrions alors des risques liés à la gestion de la mémoire. En effet, tout ce qui est référencé à partir d'un contexte statique est protégé du ramasse-miettes : il ne faudrait donc surtout pas oublier de remettre la référence statique à `null` dans `onDestroy()` car, sinon, l'objet `WeatherPlusService` resterait indéfiniment en mémoire, bien qu'il soit déconnecté du reste d'Android, et ce jusqu'à ce qu'Android décide de supprimer notre processus.

Il existe heureusement une meilleure solution : utiliser `onBind()`.Une liaison (*binding*) permet à un service d'exposer son API aux activités (ou à d'autres services) qui lui sont liés. L'essentiel de cette infrastructure est configurée pour permettre les services distants, où l'API à lier est disponible *via* IPC : un service peut donc exposer son API aux autres applications. Cependant, une liaison peut également être utile dans les situations où le service et ses clients sont tous dans la même application – c'est le scénario d'un service local.

Pour exposer le service aux autres activités *via* une liaison locale, vous devez d'abord créer une classe interne publique qui hérite de `android.os.Binder` :

```
public class LocalBinder extends Binder {
  WeatherPlusService getService() {
    return(WeatherPlusService.this);
  }
}
```

Ici, notre `Binder` n'expose qu'une seule méthode, `getService()`, qui renvoie le service lui-même. Dans un scénario de service distant, ceci ne fonctionnerait pas car les IPC nous empêchent de passer des services entre les processus. Pour les services locaux, par contre, cela fonctionne parfaitement.

Puis nous devons renvoyer cet objet `Binder` à partir de notre méthode `onBind()` :

```
@Override
public IBinder onBind(Intent intent) {
  return(binder);
}
```

Tout client qui se lie désormais à notre service sera capable d'accéder à l'objet service lui-même et d'appeler ses méthodes. Nous y reviendrons en détail au chapitre suivant.

Destinée du manifeste

Enfin, vous devez ajouter le service à votre fichier `AndroidManifest.xml` pour qu'il soit reconnu comme un service utilisable. Il suffit pour cela d'ajouter un élément `service` comme fils de l'élément `application` en utilisant l'attribut `android:name` pour désigner la classe du service.

Voici, par exemple, le fichier `AndroidManifest.xml` du projet `WeatherPlusService` :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.commonsware.android.service">
  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
  <application android:label="@string/app_name">
    <activity android:name=".WeatherPlus" android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
```

```
            <category android:name="android.intent.category.LAUNCHER" />
          </intent-filter>
        </activity>
        <service android:name=".WeatherPlusService" />
      </application>
    </manifest>
```

La classe du service étant dans le même espace de noms que tout ce qui se trouve dans cette application, vous pouvez utiliser la notation pointée simplifiée (`".WeatherPlusService"`) pour désigner votre classe.

Si vous voulez exiger certaines permissions pour lancer ou lier le service, ajoutez un attribut `android:permission` précisant la permission demandée – voir Chapitre 28 pour plus de détails.

Sauter la clôture

Parfois, le service doit prévenir de façon asynchrone une activité d'un certain événement.

La théorie derrière l'implémentation de `WeatherPlusService`, par exemple, est que le service est prévenu lorsque le terminal (ou l'émulateur) change d'emplacement. Le service appelle alors le service web et produit une nouvelle page de prévisions. Puis il doit prévenir l'activité que ces nouvelles prévisions sont disponibles, afin qu'elle puisse les charger et les afficher.

Pour interagir de cette façon avec les composants, deux possibilités sont à votre disposition : les méthodes de rappel et les intentions diffusées.

Notez que, si votre service doit simplement alerter l'utilisateur d'un certain événement, vous pouvez utiliser une notification (voir Chapitre 31), ce qui est le moyen le plus classique pour gérer ce type de scénario.

Méthodes de rappel

Une activité pouvant travailler directement avec un service local, elle peut fournir une sorte d'objet "écouteur" au service, que ce dernier pourra ensuite appeler lorsque cela est nécessaire. Pour que ceci fonctionne, vous devez :

- définir une interface Java pour cet objet écouteur ;
- donner au service une API publique pour enregistrer et supprimer des écouteurs ;
- faire en sorte que le service utilise ces écouteurs au bon moment, pour prévenir ceux qui ont enregistré l'écouteur d'un événement donné ;

- faire en sorte que l'activité enregistre et supprime un écouteur en fonction de ses besoins ;
- faire en sorte que l'activité réponde correctement aux événements gérés par les écouteurs.

La plus grande difficulté consiste à s'assurer que l'activité inactive les écouteurs lorsqu'elle se termine. Les objets écouteurs connaissent généralement leur activité, soit explicitement (*via* un membre) soit implicitement (en étant implémentés comme une classe interne). Si le service repose sur des objets écouteurs qui ont été inactivés, les activités correspondantes resteront en mémoire même si elles ne sont plus utilisées par Android, ce qui crée donc une fuite mémoire importante. Vous pouvez utiliser des `WeakReference`, des `SoftReference` ou des constructions similaires pour garantir que les écouteurs enregistrés par une activité pour votre service ne maintiendront pas cette dernière en mémoire lorsqu'elle est supprimée.

Intentions diffusées

Une autre approche que nous avions déjà mentionnée au Chapitre 17 consiste à faire en sorte que le service lance une intention diffusée qui pourra être capturée par l'activité – en supposant que cette dernière existe encore et ne soit pas en pause. Nous examinerons le côté client de cet échange au prochain chapitre mais, pour le moment, intéressons-nous à la façon dont un service peut envoyer une telle intention.

L'implémentation de haut niveau du flux est empaquetée dans `FetchForecastTask`, une implémentation d'`AsyncTask` qui permet de déplacer l'accès Internet vers un thread en arrière-plan :

```
class FetchForecastTask extends AsyncTask<Location, Void, Void> {
  @Override
  protected Void doInBackground(Location... locs) {
    Location loc=locs[0];
    String url=String.format(format, loc.getLatitude(),
                             loc.getLongitude());
    HttpGet getMethod=new HttpGet(url);

    try {
      ResponseHandler<String> responseHandler=new BasicResponseHandler();
      String responseBody=client.execute(getMethod, responseHandler);
      String page=generatePage(buildForecasts(responseBody));

      synchronized(this) {
       forecast=page;
      }

      sendBroadcast(broadcast);
    }
```

```
      catch (Throwable t) {
        android.util.Log.e("WeatherPlus",
                           "Exception dans updateForecast()", t);
      }
      return(null);
    }

  @Override
    protected void onProgressUpdate(Void... inutilise) {
      // Inutile ici
    }

    @Override
    protected void onPostExecute(Void inutilise) {
      // Inutile ici
    }
  }
```

L'essentiel de ce code ressemble au code du projet `Weather` original – il effectue la requête HTTP, convertit sa réponse en un ensemble d'objets `Forecast` qu'il transforme ensuite en page web. La première différence, outre l'introduction d'`AsyncTask`, est que la page web est simplement mise en cache dans le service car celui-ci ne peut pas directement la placer dans le `WebView` de l'activité. La seconde est que l'on appelle `sendBroadcast()`, qui prend une intention en paramètre et l'envoie à toutes les parties concernées. Cette intention a été déclarée dans le prologue de la classe :

```
private Intent broadcast=new Intent(BROADCAST_ACTION);
```

Ici, `BROADCAST_ACTION` est simplement une chaîne statique dont la valeur distingue cette intention de toutes les autres :

```
public static final String BROADCAST_ACTION =
          "com.commonsware.android.service.ForecastUpdateEvent";
```

30

Appel d'un service

Les services peuvent être utilisés par n'importe quel composant de l'application capable d'attendre pendant un certain temps. Ceci inclut les activités, les fournisseurs de contenu et les autres services. En revanche, cela ne comprend pas les récepteurs d'intention purs (les écouteurs qui ne font pas partie d'une activité) car ils sont automatiquement supprimés après le traitement d'une intention.

Pour utiliser un service local, vous devez le lancer, obtenir un accès à l'objet service, puis appeler ses méthodes. Vous pouvez ensuite arrêter le service lorsque vous n'en avez plus besoin ou, éventuellement, le laisser s'arrêter de lui-même. Dans ce chapitre, nous étudierons la partie cliente de l'application `Service/WeatherPlus`. L'activité `WeatherPlus` ressemble énormément à l'application `Weather` originale – comme le montre la Figure 30.1, il s'agit simplement d'une page web qui affiche les prévisions météorologiques.

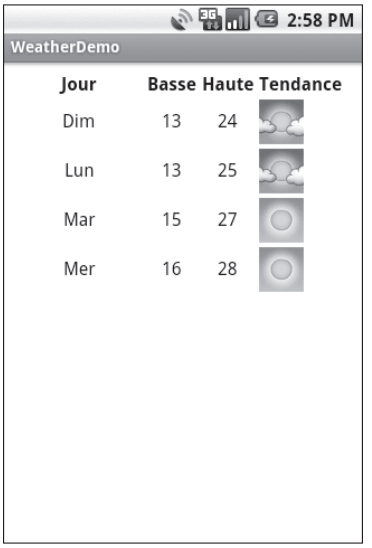


Figure 30.1
Le client du service `WeatherPlus`.

Transmission manuelle

Pour démarrer un service, une première approche consiste à appeler `startService()` en lui passant l'intention qui indique le service à lancer (là encore, le plus simple consiste à préciser la classe du service s'il s'agit de votre propre service). Inversement, on l'arrête par un appel à la méthode `stopService()`, à laquelle on passe l'intention fournie à l'appel `startService()` correspondant.

Lorsque le service est lancé, on doit communiquer avec lui. Cette communication peut être exclusivement réalisée *via* les "extras" que l'on a fournis dans l'intention ou, s'il s'agit d'un service local qui offre un singleton, utiliser ce dernier.

Si vous avez implémenté `onBind()` comme on l'a montré au chapitre précédent, vous pouvez accéder au service par un autre moyen : en appelant la méthode `bindService()`. Lorsqu'une activité se lie à un service, elle demande essentiellement à pouvoir accéder à l'API publique de ce service *via* le *binder* qui a été renvoyé par la méthode `onBind()`. L'activité peut également demander à Android, à l'aide de l'indicateur `BIND_AUTO_CREATE`, de lancer automatiquement le service si ce dernier n'a pas déjà été démarré.

Pour utiliser cette technique avec nos classes `WeatherPlus` et `WeatherPlusService`, nous devons d'abord appeler `bindService()` à partir de `onCreate()` :

```
@Override
public void onCreate(Bundle savedInstanceState) {
```

```
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    browser = (WebView)findViewById(R.id.webkit);
    bindService(new Intent(this, WeatherPlusService.class),
                onService, Context.BIND_AUTO_CREATE);
  }
```

Cet appel à `bindService()` utilise un objet de rappel `onService`, qui est une instance de `ServiceConnection` :

```
  private ServiceConnection onService=new ServiceConnection() {
    public void onServiceConnected(ComponentName className, IBinder rawBinder) {
      appService=((WeatherPlusService.LocalBinder)rawBinder).getService();
    }

    public void onServiceDisconnected(ComponentName className) {
      appService=null;
    }
  };
```

Notre objet `onService` sera appelé avec `onServiceConnected()` dès le lancement de `WeatherPlusService`. Nous passons un objet `IBinder`, qui est un descripteur opaque représentant le service et que nous pouvons utiliser pour obtenir le `LocalBinder` exposé par `WeatherPlusService`. Grâce à ce `LocalBinder`, on récupère l'objet `WeatherPlusService` lui-même, que l'on stocke dans un membre privé :

```
  private WeatherPlusService appService = null;
```

Nous pouvons ensuite appeler les méthodes de `WeatherPlusService` et notamment obtenir la page des prévisions :

```
  private void updateForecast() {
    try {
      String page=appService.getForecastPage();

      browser.loadDataWithBaseURL(null, page, "text/html",
                                  "UTF-8", null);
    }
    catch (final Throwable t) {
      goBlooey(t);
    }
  }
```

Nous devons également appeler `unbindService()` à partir d'`onDestroy()` pour libérer la liaison avec `WeatherPlusService` :

```
@Override
public void onDestroy() {
  super.onDestroy();

  unbindService(onService);
}
```

Si aucun autre client n'est lié au service, Android l'éteindra également pour libérer sa mémoire – il n'est donc pas nécessaire d'appeler explicitement `stopService()` puisque Android le fera pour nous lors de la suppression de la liaison.

Bien que cette approche nécessite un peu plus de code que la simple utilisation d'un singleton pour l'objet service, elle est moins sujette aux fuites de mémoire.

Récapitulons :

- Pour lancer un service, utilisez `bindService()` avec l'option `BIND_AUTO_CREATE` si vous voulez communiquer en utilisant le mécanisme de liaison ; `startService()` sinon.
- Pour arrêter un service, faites l'inverse : utilisez `unbindService()` ou `stopService()`.

Une autre possibilité pour arrêter un service consiste à faire en sorte que le service appelle sa méthode `stopSelf()`. Si vous utilisez `startService()` pour lancer un service et que vous effectuiez un traitement dans un thread en arrière-plan, par exemple, cette approche permettra au service de se terminer de lui-même lorsque le thread se termine.

Capture de l'intention

Au chapitre précédent, nous avons vu comment le service diffusait une intention pour signaler à l'activité `WeatherPlus` qu'un mouvement du terminal avait provoqué une modification des prévisions. Nous pouvons maintenant étudier la façon dont l'activité reçoit et utilise cette intention.

Voici les implémentations d'`onResume()` et d'`onPause()` de `WeatherPlus` :

```
@Override
public void onResume() {
  super.onResume();

  registerReceiver(receiver,
        new IntentFilter(WeatherPlusService.BROADCAST_ACTION));
}
```

```
@Override
public void onPause() {
  super.onPause();

  unregisterReceiver(receiver);
}
```

Dans `onResume()`, nous enregistrons un `BroadcastReceiver` statique pour recevoir les intentions qui correspondent à l'action déclarée par le service. Dans `onPause()`, nous désactivons ce récepteur car nous ne recevrons plus ces intentions pendant que nous sommes en pause.

Le `BroadcastReceiver`, de son côté, met simplement les prévisions à jour :

```
private BroadcastReceiver receiver=new BroadcastReceiver() {
  public void onReceive(Context context, Intent intent) {
    updateForecast();
  }
```

31

Alerter les utilisateurs avec des notifications

Les messages qui surgissent, les icônes et les "bulles" qui leur sont associées, les icônes qui bondissent dans la barre d'état, etc. sont utilisés par les programmes pour attirer votre attention, et parfois pour de bonnes raisons.

Votre téléphone vous alerte aussi probablement pour d'autres motifs que la réception d'un appel : batterie faible, alarme programmée, notifications de rendez-vous, réception de SMS ou de courrier électronique, etc.

Il n'est donc pas étonnant qu'Android dispose d'un framework complet pour gérer ce genre d'événements, désignés sous le terme de *notifications*.

Types d'avertissements

Un service qui s'exécute en arrière-plan doit pouvoir attirer l'attention des utilisateurs lorsqu'un événement survient – la réception d'un courrier, par exemple. En outre, le service doit également diriger l'utilisateur vers une activité lui permettant d'agir en réponse à cet

événement – lire le message reçu, par exemple. Pour ce type d'action, Android fournit des icônes dans la barre d'état, des avertissements lumineux et d'autres indicateurs que l'on désigne globalement par le terme de notifications.

Votre téléphone actuel peut posséder ce type d'icône pour indiquer le niveau de charge de la batterie, la force du signal, l'activation de Bluetooth, etc. Avec Android, les applications peuvent ajouter leurs propres icônes dans la barre d'état et faire en sorte qu'elles n'apparaissent que lorsque cela est nécessaire (lorsqu'un message est arrivé, par exemple).

Vous pouvez lancer des notifications *via* le `NotificationManager`, qui est un service du système. Pour l'utiliser, vous devez obtenir l'objet service *via* un appel à la méthode `getSystemService` (`NOTIFICATION_SERVICE`) de votre activité.

Le `NotificationManager` vous offre trois méthodes : une pour avertir (`notify()`) et deux pour arrêter d'avertir (`cancel()` et `cancelAll()`).

La méthode `notify()` prend en paramètre un objet `Notification`, qui est une structure décrivant la forme que doit prendre l'avertissement. Les possibilités de cet objet sont décrites dans les sections qui suivent.

Notifications matérielles

Vous pouvez faire clignoter les LED du terminal en mettant `lights` à `true` et en précisant leur couleur (sous la forme d'une valeur `#ARGB` dans `ledARGB`). Vous pouvez également préciser le type de clignotement (en indiquant les durées d'allumage et d'extinction en millisecondes dans `ledOnMS` et `ledOffMS`). Les terminaux Android feront tout leur possible pour respecter vos choix de couleurs, mais des terminaux différents peuvent produire des couleurs différentes, voire ne pas vous donner le choix : le DEXT de Motorola, par exemple, n'a qu'une seule LED blanche et n'affichera donc que cette couleur quelle que soit celle que vous avez demandée.

Vous pouvez faire retentir un son en indiquant une `Uri` vers un contenu situé, par exemple, dans un `ContentManager` (`sound`). Considérez ce son comme une sonnerie pour votre application.

Enfin, vous pouvez faire vibrer le terminal en indiquant les alternances de la vibration (`vibrate`) en millisecondes dans un tableau de valeurs `long`. Cette vibration peut être le comportement par défaut ou vous pouvez laisser le choix à l'utilisateur, au cas où il aurait besoin d'un avertissement plus discret qu'une sonnerie. Pour utiliser cette approche, vous devrez demander la permission `VIBRATE` (voir Chapitre 28).

Icônes

Alors que les lumières, les sons et les vibrations sont destinés à attirer le regard de l'utilisateur vers son terminal, les icônes constituent l'étape suivante consistant à signaler ce qui est si important.

Pour configurer une icône pour une `Notification`, vous devez initialiser deux champs publics : `icon`, qui doit fournir l'identifiant d'une ressource `Drawable` représentant l'icône voulue, et `contentIntent`, qui indique le `PendingIntent` qui devra être déclenché lorsqu'on clique sur l'icône. Vous devez vous assurer que le `PendingIntent` sera capturé – éventuellement par le code de votre application – pour prendre les mesures nécessaires afin que l'utilisateur puisse gérer l'événement qui a déclenché la notification.

Vous pouvez également fournir un texte qui apparaîtra lorsque l'icône est placée sur la barre d'état (`tickerText`).

Pour configurer ces trois composantes, l'approche la plus simple consiste à appeler la méthode `setLatestEventInfo()`, qui enveloppe leurs initialisations dans un seul appel.

Les avertissements en action

Examinons le projet `Notifications/Notify1`, notamment sa classe `NotifyDemo` :

```
package com.commonsware.android.notify;

import android.app.Activity;
import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import java.util.Timer;
import java.util.TimerTask;

public class NotifyDemo extends Activity {
  private static final int NOTIFY_ME_ID=1337;
  private Timer timer=new Timer();
  private int count=0;

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

```
        Button btn=(Button)findViewById(R.id.notify);

    btn.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        TimerTask task=new TimerTask() {
          public void run() {
            notifyMe();
          }
        };

        timer.schedule(task, 5000);
      }
    });

    btn=(Button)findViewById(R.id.cancel);

    btn.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        NotificationManager mgr=
          (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

        mgr.cancel(NOTIFY_ME_ID);
      }
    });
  }

  private void notifyMe() {
    final NotificationManager mgr=
      (NotificationManager)getSystemService(NOTIFICATION_SERVICE);
    Notification note=new Notification(R.drawable.red_ball,
                                       "Message !!!",
                                       System.currentTimeMillis());
    PendingIntent i=PendingIntent.getActivity(this, 0,
                          new Intent(this, NotifyMessage.class),
                                             0);

    note.setLatestEventInfo(this, "Titre de la notification",
                          "Message de notification", i);
    note.number=++count;

    mgr.notify(NOTIFY_ME_ID, note);
  }
}
```

Cette activité fournit deux gros boutons, un pour lancer une notification après un délai de 5 secondes, l'autre pour annuler cette notification si elle est active. La Figure 31.1 montre l'aspect de son interface lorsqu'elle vient d'être lancée.

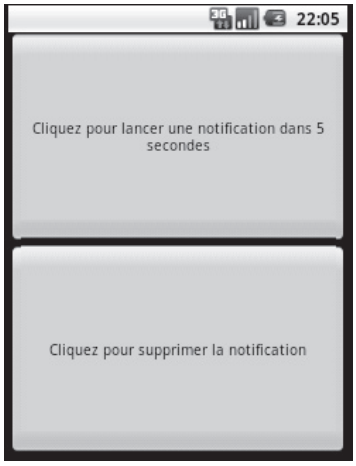


Figure 31.1
La vue principale de `NotifyDemo`.

La création de la notification dans `notifyMe()` se déroule en six étapes :

1. Obtenir l'accès à l'instance de `NotificationManager`.
2. Créer un objet `Notification` avec une icône (une boule rouge), un message qui clignote sur la barre d'état lorsque la notification est lancée et le temps associé à cet événement.
3. Créer un `PendingIntent` qui déclenchera l'affichage d'une autre activité (`Notify-Message`).
4. Utiliser `setLatestEventInfo()` pour préciser que l'on affiche un titre et un message lorsqu'on clique sur la notification et qu'on lance le `PendingIntent` lorsque l'on clique sur le message.
5. Mettre à jour le numéro associé à la notification.
6. Demander au `NotificationManager` d'afficher la notification.

Par conséquent, si l'on clique sur le bouton du haut, la boule rouge apparaîtra dans la barre de menu après un délai de 5 secondes, accompagnée brièvement du message d'état, comme le montre la Figure 31.2. Après la disparition du message, un chiffre s'affichera sur la boule rouge (initialement 1) – qui pourrait par exemple indiquer le nombre de messages non lus.

En cliquant sur la boule rouge, un tiroir apparaîtra sous la barre d'état. L'ouverture de ce tiroir montrera les notifications en suspens, dont la nôtre, comme le montre la Figure 31.3.

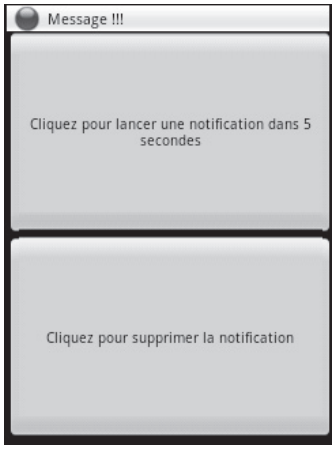


Figure 31.2
Notre notification apparaît dans la barre d'état, avec le message d'état.

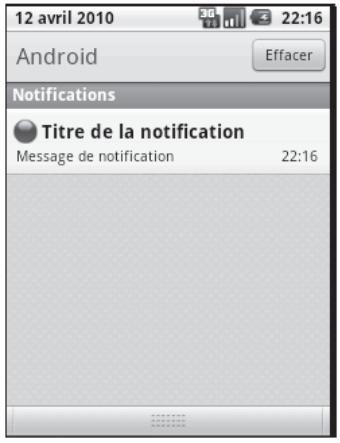


Figure 31.3
Le tiroir des notifications complètement ouvert, contenant notre notification.

En cliquant sur l'entrée de la notification dans la barre de notification, on déclenche une activité très simple se bornant à afficher un message – dans une vraie application, cette activité réaliserait un traitement tenant compte de l'événement qui est survenu (présenter à l'utilisateur les nouveaux messages de courrier, par exemple).

Si l'on clique sur le bouton Effacer du tiroir, la balle rouge disparaît de la barre d'état.

32

Accès aux services de localisation

Le GPS (*Global Positioning System*) est une fonctionnalité très appréciée des terminaux mobiles actuels car il permet de vous indiquer votre emplacement géographique à tout moment. Bien que l'utilisation la plus fréquente du GPS soit la cartographie et l'orientation, connaître votre emplacement vous ouvre de nombreux autres horizons. Vous pouvez, par exemple, mettre en place une application de chat dynamique où vos contacts sont classés selon leurs emplacements géographiques, afin de choisir ceux qui sont les plus près de vous. Vous pouvez également "géotaguer" automatiquement les articles que vous postez sur Twitter ou d'autres services similaires.

Cependant, le GPS n'est pas le seul moyen d'identifier un emplacement géographique :

- L'équivalent européen de GPS, Galileo, est encore en cours de mise au point.
- La triangulation permet de déterminer votre position en fonction de la force du signal des antennes relais proches de vous.
- La proximité des "hotspots" Wifi, dont les positions géographiques sont connues.

Les terminaux Android peuvent utiliser un ou plusieurs de ces services. En tant que développeur, vous pouvez demander au terminal de vous indiquer votre emplacement, ainsi que des détails sur les fournisseurs disponibles. Vous pouvez même simuler votre localisation avec l'émulateur pour tester les applications qui ont besoin de cette fonctionnalité.

Fournisseurs de localisation : ils savent où vous vous cachez

Les terminaux Android peuvent utiliser plusieurs moyens pour déterminer votre emplacement géographique. Certains ont une meilleure précision que d'autres ; certains vous fourniront ce service gratuitement, tandis que d'autres vous le feront payer ; certains peuvent vous donner des informations supplémentaires, comme votre altitude ou votre vitesse courante.

Android a donc abstrait tout cela en un ensemble d'objets `LocationProvider`. Votre environnement utilisera zéro ou plusieurs instances de `LocationProvider`, une par service de localisation disponible sur le terminal. Ces fournisseurs ne connaissent pas seulement votre emplacement mais possèdent également leurs propres caractéristiques – précision, prix, etc.

Vous aurez donc besoin d'un `LocationManager` contenant l'ensemble des `LocationProvider` pour savoir quel est le `LocationProvider` qui convient à votre cas particulier. Votre application devra également disposer d'une permission ; sinon les différentes API de localisation échoueront à cause d'une violation de sécurité. Selon les fournisseurs de localisation que vous voulez utiliser, vous aurez besoin des permissions `ACCESS_COARSE_LOCATION`, `ACCESS_FINE_LOCATION` ou les deux (les permissions sont décrites au Chapitre 28).

Se trouver soi-même

L'opération la plus évidente d'un fournisseur de localisation consiste à trouver votre emplacement actuel.

Pour ce faire, vous avez besoin d'un `LocationManager`, que vous obtiendrez par un appel à `getSystemService(LOCATION_SERVICE)` à partir de votre activité ou service, en transtypant le résultat pour obtenir un `LocationManager`.

L'étape suivante consiste à obtenir le nom du `LocationProvider` que vous voulez utiliser. Pour ce faire, deux possibilités s'offrent à vous :

- demander à l'utilisateur de choisir un fournisseur ;
- trouver le fournisseur qui convient le mieux en fonction d'un ensemble de critères.

Si vous choisissez la première approche, un appel à la méthode `getProviders()` du `LocationManager` vous donnera une liste de fournisseurs que vous pouvez présenter à l'utilisateur pour qu'il fasse son choix.

Vous pouvez également créer et initialiser un objet `Criteria`, en précisant ce que vous attendez d'un `LocationProvider`. Par exemple :

- `setAltitudeRequired()` pour indiquer si vous avez besoin ou non de connaître votre altitude ;
- `setAccuracy()` pour fixer un niveau de précision minimal de la position, en mètres ;
- `setCostAllowed()` pour indiquer si le fournisseur doit être gratuit ou non (c'est-à-dire s'il peut impliquer un paiement de la part de l'utilisateur du terminal).

Lorsque l'objet `Criteria` a été rempli, appelez la méthode `getBestProvider()` de votre `LocationManager` et Android passera les critères en revue pour vous donner la meilleure réponse. Tous ces critères peuvent ne pas être vérifiés – à part celui concernant le prix, ils peuvent tous être ignorés si rien ne correspond.

Pour effectuer des tests, vous pouvez également indiquer directement dans votre code Java le nom d'un `LocationProvider` (`GPS_PROVIDER`, par exemple).

Lorsque vous connaissez le nom du `LocationProvider`, vous pouvez appeler `getLastKnownPosition()` pour trouver votre dernière position. Notez, cependant, que cette "dernière position" peut être obsolète (si, par exemple, le téléphone était éteint) ou valoir `null` si aucune position n'a encore été enregistrée pour ce fournisseur. En revanche, `getLastKnownPosition()` est gratuite et ne consomme pas de ressource car le fournisseur n'a pas besoin d'être activé pour connaître cette valeur.

Cette méthode renvoie un objet `Location` qui vous indiquera la latitude et la longitude du terminal en degrés – des valeurs `double` de Java. Si le fournisseur donne d'autres informations, vous pouvez les récupérer à l'aide des méthodes suivantes :

- `hasAltitude()` indique s'il y a une valeur pour l'altitude et `getAltitude()` renvoie l'altitude en mètres.
- `hasBearing()` indique s'il y a une information d'orientation (une valeur de compas) et `getBearing()` renvoie cette valeur en degrés par rapport au vrai nord.
- `hasSpeed()` indique si la vitesse est connue et `getSpeed()` la renvoie en mètres par seconde.

Ceci dit, une approche plus fréquente pour obtenir l'objet `Location` à partir d'un `LocationProvider` consiste à s'enregistrer pour être prévenu des modifications de la position du terminal, comme on l'explique dans la section suivante.

Se déplacer

Tous les fournisseurs de localisation ne répondent pas immédiatement. GPS, par exemple, nécessite l'activation d'un signal et la réception des satellites (c'est ce que l'on appelle un "fix GPS") avant de pouvoir connaître sa position. C'est la raison pour laquelle Android ne fournit pas de méthode `getMeMyCurrentLocationNow()`. Ceci combiné avec le fait que les utilisateurs puissent vouloir que leurs mouvements soient pris en compte dans l'application, vous comprendrez pourquoi il est préférable de signaler que l'on veut connaître les modifications de la position et s'en servir pour connaître sa position courante.

Les applications `Weather` et `WeatherPlus` montrent comment être prévenu de ces mises à jour – en appelant la méthode `requestLocationUpdates()` de l'objet `LocationManager`. Cette méthode prend quatre paramètres :

- Le nom du fournisseur de localisation que vous souhaitez utiliser.
- Le temps, en millisecondes, qui doit s'écouler avant que l'on puisse obtenir une mise à jour de la position.
- Le déplacement minimal du terminal en mètres pour que l'on puisse obtenir une mise à jour de la position.
- Un `LocationListener` qui sera prévenu des événements liés à la localisation.

Voici un exemple de `LocationListener` :

```
LocationListener onLocationChange = new LocationListener() {
  public void onLocationChanged(Location location) {
    updateForecast(location);
  }

  public void onProviderDisabled(String provider) {
    // Exigée par l'interface, mais inutilisée
  }

  public void onProviderEnabled(String provider) {
    // Exigée par l'interface, mais inutilisée
  }

  public void onStatusChanged(String provider, int status,
                              Bundle extras) {
    // Exigée par l'interface, mais inutilisée
  }
};
```

Ici, nous appelons simplement `updateForecast()` en lui passant l'objet `Location` fourni à l'appel de la méthode de rappel `onLocationChanged()`. Comme on l'a vu au Chapitre 29,

l'implémentation d'`updateForecast()` construit une page web contenant les prévisions météorologiques pour l'emplacement courant et envoie un message de diffusion afin que l'activité sache qu'une mise à jour est disponible.

Lorsque l'on n'a plus besoin des mises à jour, on appelle `removeUpdates()` avec le `LocationListener` que l'on avait enregistré. Si l'on oublie de le faire, l'application continuera de recevoir des mises à jour de l'emplacement, même après que toutes les activités se seront terminées, et Android ne pourra pas récupérer la mémoire occupée par cette application.

Est-on déjà arrivé ? Est-on déjà arrivé ? Est-on déjà arrivé ?

Parfois, on veut savoir non pas où l'on se trouve ni même où l'on va, mais si l'on est là où l'on voulait aller. Il pourrait s'agir d'une destination finale ou d'une étape dans un ensemble de directions pour pouvoir indiquer le virage suivant, par exemple.

Dans ce but, `LocationManager` fournit la méthode `addProximityAlert()`, qui enregistre un `PendingIntent` qui sera déclenché lorsque le terminal se trouvera à une certaine distance d'un emplacement donné. La méthode `addProximityAlert()` attend les paramètres suivants :

- La latitude et la longitude de la position qui nous intéresse.
- Un rayon précisant la proximité avec la position pour que l'intention soit levée.
- Une durée d'enregistrement en millisecondes – passée cette période, l'enregistrement expirera automatiquement. Une valeur de –1 indique que l'enregistrement sera maintenu jusqu'à ce que vous le supprimiez manuellement *via* un appel à `removeProximityAlert()`.
- Le `PendingIntent` qu'il faudra lever lorsque le terminal se trouve dans la "zone de tir" définie par la position et le rayon.

Notez qu'il n'est pas garanti que vous receviez une intention s'il y a eu une interruption dans les services de localisation ou si le terminal n'est pas dans la zone cible pendant le temps où l'alerte de proximité est active. Si la position, par exemple, est trop proche du but et que le rayon est un peu trop réduit, le terminal peut ne faire que longer le bord de la zone cible ou y passer si rapidement que sa position ne sera pas enregistrée pendant qu'il est dans la zone.

Il vous appartient de faire en sorte qu'une activité ou un récepteur d'intention réponde à l'intention que vous avez enregistrée pour l'alerte de proximité. C'est également à vous de déterminer ce qui doit se passer lorsque cette intention arrive : configurer une notification (faire vibrer le terminal, par exemple), enregistrer l'information dans un fournisseur de contenu, poster un message sur un site web, etc.

Notez que vous recevrez l'intention à chaque fois que la position est enregistrée et que vous êtes dans la zone ciblée – pas simplement lorsque vous y entrez. Par conséquent, vous la recevrez plusieurs fois – le nombre d'occurrences dépend de la taille de la zone et de la vitesse de déplacement du terminal.

Tester... Tester...

L'émulateur d'Android ne permet pas d'obtenir un "fix GPS", de trianguler votre position à partir des antennes relais ni de déduire votre position à partir des signaux Wifi voisins. Si vous voulez simuler un terminal qui se déplace, il faut donc trouver un moyen de fournir à l'émulateur des données de localisation simulées.

Pour une raison inconnue, ce domaine a subi des changements importants au cours de l'évolution d'Android. À une époque, il était possible de fournir des données de localisation simulées à une application, ce qui était très pratique pour les tests et les démonstrations mais, malheureusement, cette possibilité a disparu à partir d'Android 1.0.

Ceci dit, DDMS (*Dalvik Debug Monitor Service*) permet de fournir ce type de données. Il s'agit d'un programme externe, séparé de l'émulateur, qui peut fournir à ce dernier des points d'emplacements ou des routes complètes dans différents formats. DDMS est décrit en détail au Chapitre 35.

33

Cartographie avec *MapView* et *MapActivity*

Google Maps est l'un des services les plus connus de Google – après le moteur de recherche, bien entendu. Avec lui, vous pouvez tout trouver, de la pizzeria la plus proche au trajet menant de Toulouse à Paris en passant par les vues détaillées des rues (*Street View*) et les images satellites.

La plupart des terminaux Android intègrent Google Maps, ce qui n'est guère surprenant. Pour ceux qui en disposent, cette activité de cartographie est directement disponible à partir du menu principal. Ce qui nous intéresse surtout ici est que les développeurs ont à leur disposition les classes `MapView` et `MapActivity` pour intégrer des cartes géographiques dans leurs applications. Grâce à elles, ils peuvent non seulement contrôler le niveau du zoom, permettre aux utilisateurs de faire défiler la carte, mais également utiliser les services de localisation (voir Chapitre 32) pour marquer l'emplacement du terminal et indiquer son déplacement.

Heureusement, cette intégration est assez simple et vous pouvez exploiter toute sa puissance si vous le souhaitez.

Termes d'utilisation

Pour être intégré dans des applications tierces, Google Maps nécessite l'acceptation d'un assez grand nombre de termes juridiques. Parmi ceux-ci se trouvent des clauses que vous trouverez peut-être insupportables. Si vous décidez d'utiliser Google Maps, prenez soin de bien lire tous ces termes afin d'être sûr que l'utilisation que vous comptez en faire ne les viole pas. Nous vous conseillons fortement de demander l'avis d'un conseiller juridique en cas de doute.

En outre, ne délaissez pas les autres possibilités de cartographie qui reposent sur d'autres sources de données géographiques, comme OpenStreetMap[1].

Empilements

À partir d'Android 1.5, Google Maps ne fait plus partie du SDK à proprement parler mais a été déplacé dans les API supplémentaires de Google, qui sont des extensions au SDK de base. Ce système d'extension fournit des points d'entrée aux autres sous-systèmes qui peuvent se trouver sur certains terminaux mais pas sur d'autres.

Info

En réalité, Google Maps ne fait pas partie du projet open-source Android et il existera nécessairement des terminaux qui n'en disposeront pas à cause des problèmes de licence. À l'heure actuelle, par exemple, la tablette Android Archos 5 ne dispose pas de Google Maps.

Dans l'ensemble, le fait que Google Maps soit une extension n'affectera pas votre développement habituel à condition de ne pas oublier les points suivants :

- Vous devrez créer votre projet pour qu'il utilise une cible adaptée afin d'être sûr que les API de Google Maps soient disponibles.
- Pour tester l'intégration de Google Maps, vous aurez également besoin d'un AVD qui supporte les API de Google Maps.

1. **http://www.openstreetmap.org/**.

Les composants essentiels

Pour insérer une carte géographique dans une application, le plus simple consiste à créer une sous-classe de `MapActivity`. Comme `ListActivity`, qui enveloppe une partie des détails cachés derrière une activité dominée par une `ListView`, `MapActivity` gère une partie de la configuration d'une activité dominée par une `MapView`.

Dans le fichier de layout de la sous-classe de `MapActivity`, vous devez ajouter un élément qui, actuellement, s'appelle `com.google.android.maps.MapView`. Il s'agit ici d'un nom totalement développé qui ajoute le nom de paquetage complet au nom de la classe (cette notation est nécessaire car `MapView` ne se trouve pas dans l'espace de noms `com.google.android.widget`). Vous pouvez donner la valeur que vous souhaitez à l'attribut `android:id` du widget `MapView` et gérer tous les détails lui permettant de s'afficher correctement à côté des autres widgets.

Vous devez cependant préciser les attributs suivants :

- `android:apiKey`. Dans une version de l'application en production, cet attribut doit contenir une clé de l'API Google Maps (voir plus loin).
- `android:clickable = "true"`. Si vous voulez que les utilisateurs puissent cliquer sur la carte et la faire défiler.

Voici, par exemple, le contenu du fichier layout principal de l'application `Maps/NooYawk` :

```
<?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="fill_parent">
  <com.google.android.maps.MapView android:id="@+id/map"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:apiKey=<VOTRE CLÉ API>
    android:clickable="true" />
</RelativeLayout>
```

Nous présenterons ce mystérieux `apiKey` plus bas dans ce chapitre.

Vous devez également ajouter deux informations supplémentaires à votre fichier `AndroidManifest.xml` :

- Les permissions `INTERNET` et `ACCESS_COARSE_LOCATION` (cette dernière est utilisée par la classe `MyLocationOverlay`, que nous décrirons plus loin).
- Dans l'élément `<application>`, ajoutez un élément `<uses-library>` avec l'attribut `android:name = "com.google.android.maps"` pour indiquer que vous utilisez l'une des API facultatives d'Android.

Voici le fichier AndroidManifest.xml du projet NooYawk :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.maps">
  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />

  <application android:label="@string/app_name"
    android:icon="@drawable/cw">
    <uses-library android:name="com.google.android.maps" />
    <activity android:name=".NooYawk" android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Avec la sous-classe de MapActivity, c'est à peu près tout ce dont vous avez besoin pour débuter. Si vous ne faites rien d'autre, que vous compiliez ce projet et que vous l'installiez dans l'émulateur, vous obtiendrez une belle carte du monde. Notez, cependant, que MapActivity est une classe abstraite et que vous devez donc implémenter la méthode isRouteDisplayed() pour préciser si vous fournissez ou non une gestion des itinéraires.

En théorie, l'utilisateur doit pouvoir faire défiler la carte en utilisant le pad directionnel. Cependant, ce n'est pas très pratique lorsque l'on a le monde entier dans sa main...

Une carte du monde n'étant pas très utile en elle-même, nous devons lui ajouter quelques fonctionnalités.

Testez votre contrôle

Pour trouver votre widget MapView, il suffit, comme d'habitude, d'appeler la méthode findViewById(). Le widget lui-même fournit la méthode getMapController(). Entre le MapView et le MapController, vous disposez d'un bon nombre de possibilités pour déterminer ce qu'affiche la carte et la façon dont elle se comporte ; les sections suivantes présentent le zoom et le centrage, qui sont sûrement celles que vous utiliserez le plus.

Zoom

La carte du monde avec laquelle vous démarrez est plutôt vaste. Sur un téléphone, on préfère généralement consulter une carte ayant une étendue plus réduite – quelques pâtés de maisons, par exemple.

Vous pouvez contrôler directement le niveau du zoom grâce à la méthode `setZoom()` de `MapController`. Celle-ci attend un paramètre entier représentant le niveau du zoom, où `1` représente la vue du monde entier et `21`, le plus fort grossissement que vous pouvez obtenir. Chaque niveau double la résolution effective par rapport au niveau précédent : au niveau `1`, l'équateur fait 256 pixels de large et il en fait 268 435 456 au niveau `21`. L'écran du téléphone n'ayant sûrement pas autant de pixels ; l'utilisateur ne verra donc qu'une petite partie de la carte centrée sur un endroit du globe. Le niveau `16` montrera plusieurs pâtés de maisons dans chaque dimension et constitue généralement un bon point de départ pour vos essais.

Si vous souhaitez que les utilisateurs aient le droit de changer le niveau du zoom, utilisez l'appel `setBuiltInZoomControls(true)` : l'utilisateur pourra alors utiliser les contrôles de zoom qui se trouvent en bas de la carte.

Centrage

Généralement, quel que soit le niveau du zoom, vous voudrez contrôler ce qui est affiché sur la carte : la position courante de l'utilisateur ou un emplacement sauvegardé avec d'autres données de votre activité, par exemple. Pour changer la position de la carte, appelez la méthode `setCenter()` de `MapController`.

Cette méthode prend un objet `GeoPoint` en paramètre. Un `GeoPoint` représente un emplacement exprimé par une latitude et une longitude. En réalité, il les stocke sous la forme d'entiers en multipliant leurs vraies valeurs par `1E6`, ce qui permet d'économiser un peu de mémoire par rapport à des `float` ou à des `double` et d'accélérer un peu la conversion d'un `GeoPoint` en position sur la carte. En revanche, vous ne devez pas oublier ce facteur de `1E6`.

Terrain accidenté

Tout comme sur votre ordinateur de bureau, vous pouvez afficher les images satellites avec Google Maps et Android.

`MapView` offre la méthode `toggleSatellite()`, qui, comme son nom l'indique, permet d'activer ou de désactiver la vue satellite de la surface présentée sur la carte. Vous pouvez faire en sorte de laisser à l'utilisateur le soin de faire ce choix à partir d'un menu ou, comme dans `NooYawk`, *via* des touches :

```
@Override
 public boolean onKeyDown(int keyCode, KeyEvent event) {
   if (keyCode == KeyEvent.KEYCODE_S) {
     map.setSatellite(!map.isSatellite());
     return(true);
```

```
    }
    else if (keyCode == KeyEvent.KEYCODE_Z) {
      map.displayZoomControls(true);
      return(true);
    }

    return(super.onKeyDown(keyCode, event));
  }
```

Couches sur couches

Si vous avez déjà utilisé la version complète de Google Maps, vous avez sûrement déjà vu que l'on pouvait déposer des choses sur la carte elle-même : les "repères", par exemple, qui indiquent les emplacements des points d'intérêt proches de la position que vous avez demandée. En termes de carte – et également pour la plupart des éditeurs graphiques sérieux –, ces repères sont placés sur une couche distincte de celle de la carte elle-même et ce que vous voyez au final est la superposition de ces deux couches.

Android permet de créer de telles couches, afin de marquer les cartes en fonction des choix de l'utilisateur et des besoins de votre application. `NooYawk`, par exemple, utilise une couche pour montrer les emplacements des immeubles sélectionnés dans Manhattan.

Classes *Overlay*

Toute couche ajoutée à votre carte doit être implémentée comme une sous-classe d'`Overlay`. Si vous voulez simplement ajouter des repères, vous pouvez utiliser la sous-classe `ItemizedOverlay`, qui vous simplifiera la tâche.

Pour attacher une couche à votre carte, il suffit d'appeler la méthode `getOverlays()` de votre objet `MapView` et d'ajouter votre instance d'`Overlay` avec `add()` :

```
marker.setBounds(0, 0, marker.getIntrinsicWidth(),
                       marker.getIntrinsicHeight());

map.getOverlays().add(new SitesOverlay(marker));
```

Nous expliquerons un peu plus loin le rôle de `marker`.

Affichage d'*ItemizedOverlay*

Comme son nom l'indique, `ItemizedOverlay` permet de fournir une liste de points d'intérêt (des instances d'`OverlayItem`) pour les afficher sur la carte. La couche gère ensuite l'essentiel du dessin pour vous, mais vous devez toutefois effectuer les opérations suivantes :

1. Dérivez votre sous-classe (`SitesOverlay`, dans notre exemple) d'`ItemizedOverlay<OverlayItem>`.
2. Dans le constructeur, mettez en place la liste des instances `OverlayItem` et appelez `populate()` lorsqu'elles sont prêtes à être utilisées par la couche.
3. Implémentez `size()` pour qu'elle renvoie le nombre d'éléments qui devront être gérés par la couche.
4. Redéfinissez `createItem()` pour qu'elle renvoie l'instance `OverlayItem` correspondant à l'indice qui lui est passé en paramètre.
5. Lors de l'instanciation de la sous-classe d'`ItemizedOverlay`, fournissez-lui un objet `Drawable` représentant l'icône par défaut de chaque élément (une épinglette, par exemple).

Le `marker` que l'on passe au constructeur de `NooYawk` est le `Drawable` utilisé dans la dernière étape – il affiche une épinglette.

Vous pouvez également redéfinir `draw()` pour mieux gérer l'ombre de vos marqueurs. Bien que la carte fournisse une ombre, il peut être utile de l'aider un peu en lui indiquant où se trouve le bas de l'icône, afin qu'elle puisse en tenir compte pour l'ombrage.

Voici, par exemple, le code de la classe `SitesOverlay` :

```
private class SitesOverlay extends ItemizedOverlay<OverlayItem> {
  private List<OverlayItem> items=new ArrayList<OverlayItem>();
  private Drawable marker=null;

  public SitesOverlay(Drawable marker) {
    super(marker);
    this.marker=marker;

    items.add(new OverlayItem(getPoint(40.748963847316034,
                                       -73.96807193756104),
                              "UN", "Nations Unies"));
    items.add(new OverlayItem(getPoint(40.76866299974387,
                                       -73.98268461227417),
                              "Lincoln Center",
                              "La maison du Jazz"));
    items.add(new OverlayItem(getPoint(40.765136435316755,
                                       -73.97989511489868),
                              "Carnegie Hall",
            "Entraînez-vous avant d'y jouer !"));
```

```
    items.add(new OverlayItem(getPoint(40.70686417491799,
                                       -74.01572942733765),
                              "The Downtown Club",
                   "Le lieu d'origine du trophée Heisman"));

    populate();
  }

  @Override
  protected OverlayItem createItem(int i) {
    return(items.get(i));
  }

  @Override
  public void draw(Canvas canvas, MapView mapView,
                   boolean shadow) {
    super.draw(canvas, mapView, shadow);
    boundCenterBottom(marker);
  }

  @Override
  protected boolean onTap(int i) {
    Toast.makeText(NooYawk.this,
                   items.get(i).getSnippet(),
                   Toast.LENGTH_SHORT).show();
    return(true);
  }

  @Override
  public int size() {
    return(items.size());
  }
}
```

Gestion de l'écran tactile

Une sous-classe d'`Overlay` peut également implémenter `onTap()` pour être prévenue lorsque l'utilisateur touche la carte afin que la couche ajuste ce qu'elle affiche. Dans Google Maps, cliquer sur une épinglette fait surgir une bulle d'information consacrée à l'emplacement marqué, par exemple : grâce à `onTap()`, vous pouvez obtenir le même résultat avec Android.

La méthode `onTap()` d'`ItemizedOverlay` prend en paramètre l'indice de l'objet `OverlayItem` sur lequel on a cliqué. C'est ensuite à vous de traiter cet événement.

Dans le cas de la classe `SitesOverlay` que nous venons de présenter, le code d'`onTap()` est le suivant :

```
@Override
protected boolean onTap(int i) {
  Toast.makeText(NooYawk.this,
                 items.get(i).getSnippet(),
                 Toast.LENGTH_SHORT).show();
  return(true);
}
```

Ici, on lève simplement un `Toast` contenant le texte associé à l'`OverlayItem` et l'on renvoie `true` pour indiquer que l'on a géré le toucher de cet objet.

Moi et *MyLocationOverlay*

Android dispose d'une couche intégrée permettant de gérer deux scénarios classiques :

- l'affichage de votre position sur la carte, en fonction du GPS ou d'un autre fournisseur de localisation ;
- l'affichage de la direction vers laquelle vous vous dirigez, en fonction de la boussole intégrée lorsqu'elle est disponible.

Il vous suffit pour cela de créer une instance de `MyLocationOverlay`, de l'ajouter à la liste des couches de votre `MapView` et d'activer et de désactiver ces fonctionnalités aux moments opportuns.

La notion de "moments opportuns" est liée à l'économie de la batterie. Comme il n'y a aucune raison de mettre à jour des emplacements ou des directions lorsque l'activité est en pause, il est conseillé d'activer ces fonctionnalités dans `onResume()` et de les désactiver dans `onPause()`.

Pour que `NooYawk` affiche une boussole dans `MyLocationOverlay`, par exemple, nous devons d'abord créer la couche et l'ajouter à la liste des couches :

```
me=new MyLocationOverlay(this, map);
map.getOverlays().add(me);
```

Puis nous activons et désactivons cette boussole lorsque cela est nécessaire :

```
@Override
public void onResume() {
```

```
    super.onResume();
    me.enableCompass();
  }
  @Override
  public void onPause() {
    super.onPause();
    me.disableCompass();
  }
```

La clé de tout

Si vous compilez le projet `NooYawk` et que vous l'installiez dans votre émulateur, vous verrez sûrement un écran montrant une grille et deux épinglettes, mais pas de carte.

La raison en est que la clé de l'API dans le code source n'est pas valide pour votre machine de développement. Vous devez donc produire votre propre clé pour l'utiliser avec votre application.

Le site web d'Android[1] donne toutes les instructions nécessaires pour produire ces clés, que ce soit pour le développement ou pour la production. Pour rester brefs, nous nous intéresserons ici au cas particulier de l'exécution de `NooYawk` dans votre émulateur. Vous devez effectuer les étapes suivantes :

1. Allez sur la page d'inscription pour la clé de l'API et lisez les termes d'utilisation.
2. Relisez ces termes et soyez absolument sûr que vous les approuvez.
3. Recherchez la signature MD5 du certificat utilisé pour signer vos applications en mode debug (voir ci-après).
4. Sur la page d'inscription pour la clé de l'API, collez cette signature MD5 et envoyez le formulaire.
5. Sur la page de réponse, copiez la clé de l'API et collez-la dans la valeur de l'attribut `android:apiKey` du layout de votre `MapView`.

La partie la plus compliquée consiste à trouver la signature MD5 du certificat utilisé pour signer vos applications en mode debug... et une bonne partie de cette complexité consiste à comprendre le concept.

Toutes les applications Android sont signées à l'aide d'une signature numérique produite à partir d'un certificat. Vous recevez automatiquement un certificat de débogage lorsque vous installez le SDK et il faut suivre un autre processus pour créer un certificat autosigné

1. **http://code.google.com/android/toolbox/apis/mapkey.html**.

utilisable avec vos applications en production. Ce processus nécessite d'utiliser les outils `keytool` et `jarsigner` de Java. Pour obtenir votre clé d'API, vous n'avez besoin que de `keytool`.

Si vous utilisez OS X ou Linux, faites la commande suivante pour obtenir la signature MD5 de votre certificat de débogage :

```
keytool -list -alias androiddebugkey -keystore ~/.android/debug.keystore
➥-storepass android -keypass android
```

Sur les autres plates-formes de développement, vous devrez remplacer la valeur de `-keystore` par l'emplacement sur votre machine et votre compte utilisateur :

Windows XP : `C:\Documents` et `Settings\<utilisateur>\.android\debug.keystore`.

Windows Vista et Windows 7 : `C:\Users\<utilisateur>\.android\debug.keystore` (où `<utilisateur>` est le nom de votre compte).

La seconde ligne du résultat qui s'affiche contient votre signature MD5, qui est une suite de paires de chiffres hexadécimaux séparées par des caractères deux-points.

34

Gestion des appels téléphoniques

La plupart des terminaux Android, si ce n'est tous, sont des téléphones. Leurs utilisateurs s'attendent donc à pouvoir téléphoner et recevoir des appels et, si vous le souhaitez, vous pouvez les y aider :

- Vous pourriez créer une interface pour une application de gestion des ventes (à la Salesforce.com) en offrant la possibilité d'appeler les vendeurs d'un simple clic, sans que l'utilisateur soit obligé de mémoriser ses contacts à la fois dans l'application et dans son répertoire téléphonique.
- Vous pourriez développer une application de réseau social avec une liste de numéros de téléphone qui évolue sans cesse : au lieu de devoir constamment "synchroniser" ses contacts avec ceux du téléphone, l'utilisateur pourrait les appeler directement à partir de cette application.

- Vous pourriez créer une interface personnalisée pour le système de contacts existants, éventuellement pour que les utilisateurs à mobilité réduite (comme les personnes âgées) puissent disposer de gros boutons pour faciliter la composition des appels.

Quoi qu'il en soit, Android vous permet de manipuler le téléphone comme n'importe quelle autre composante du système.

Le Manager

Pour tirer le meilleur parti de l'API de téléphonie, utilisez la classe `TelephonyManager`, qui permet notamment de :

- déterminer si le téléphone est en cours d'utilisation, *via* sa méthode `getCallState()`, qui renvoie les valeurs `CALL_STATE_IDLE` (téléphone non utilisé), `CALL_STATE_RINGING` (appel en cours de connexion) et `CALL_STATE_OFFHOOK` (appel en cours) ;
- trouver l'identifiant de la carte SIM avec `getSubscriberId()` ;
- connaître le type du téléphone (GSM, par exemple) avec `getPhoneType()` ou celui de la connexion (comme GPRS, EDGE) avec `getNetworkType()`.

Appeler

Pour effectuer un appel à partir d'une application, en utilisant par exemple un numéro que vous avez obtenu par votre propre service web, créez une intention `ACTION_DIAL` avec une `Uri` de la forme `tel:`*`NNNNN`* (où *NNNNN* est le numéro de téléphone à appeler) et utilisez cette intention avec `startActivity()`. Cela ne lancera pas l'appel mais activera l'activité du combiné, à partir duquel l'utilisateur pourra alors appuyer sur un bouton pour effectuer l'appel.

Voici, par exemple, un fichier de disposition simple mais efficace, extrait du projet `Phone/Dialer` :

```
<?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"
    >
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
```

```
      >
      <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Numéro : "
        />
      <EditText android:id="@+id/number"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:cursorVisible="true"
        android:editable="true"
        android:singleLine="true"
      />
    </LinearLayout>
    <Button android:id="@+id/dial"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:text="Appeler !"
    />
  </LinearLayout>
```

Nous utilisons simplement un champ de saisie pour entrer un numéro de téléphone et un bouton pour appeler ce numéro.

Le code Java se contente de lancer le combiné en utilisant le numéro saisi dans le champ :

```
package com.commonsware.android.dialer;
import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
public class DialerDemo extends Activity {
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);

    final EditText number=(EditText)findViewById(R.id.number);
    Button dial=(Button)findViewById(R.id.dial);

    dial.setOnClickListener(new Button.OnClickListener() {
```

```
            public void onClick(View v) {
              String toDial="tel:" + number.getText().toString();

              startActivity(new Intent(Intent.ACTION_DIAL,
                                       Uri.parse(toDial)));
            }
          });
        }
      }
```

Comme le montre la Figure 34.1, l'interface de cette activité n'est pas très impressionnante.

Figure 34.1
L'application `DialerDemo` *lors de son lancement.*

Cependant, le combiné téléphonique que l'on obtient en cliquant sur le bouton "Appeler !" est plus joli, comme le montre la Figure 34.2.

Figure 34.2
L'activité `Dialer` *d'Android lancée à partir de* `Dialer-Demo`.

35

Outils de développement

Le SDK Android n'est pas qu'une bibliothèque de classes Java et d'API, il contient également un certain nombre d'outils permettant de faciliter le développement des applications.

Nous avons surtout évoqué l'extension pour Eclipse, qui intègre le processus de développement Android dans cet IDE ainsi que les outils en ligne de commande, comme `adb`, qui permet de communiquer avec l'émulateur.

Dans ce chapitre, nous nous intéresserons aux autres outils.

Gestion hiérarchique

Android est fourni avec un outil permettant de visualiser une hiérarchie, conçu pour vous aider à consulter vos layouts tels qu'ils sont vus par une activité en cours d'exécution dans un émulateur. Vous pouvez ainsi savoir l'espace qu'occupe un widget ou trouver où se cache un widget particulier qui n'apparaît pas à l'écran.

Pour utiliser cet outil, vous devez d'abord lancer l'émulateur, installer votre application, lancer l'activité et naviguer vers l'endroit que vous souhaitez examiner. Notez que vous ne pouvez pas utiliser ce visualisateur avec un vrai terminal Android, comme le G1. Comme le

montre la Figure 35.1, nous utiliserons à titre d'exemple l'application `ReadWriteFileDemo` que nous avions présentée au Chapitre 23.

Pour lancer le visualisateur, utilisez le programme `hierarchyviewer` qui se trouve dans le répertoire `tools/` de votre installation du SDK. Vous obtiendrez alors la fenêtre présentée à la Figure 35.2.

Figure 35.1
L'application `ReadWriteFileDemo`.

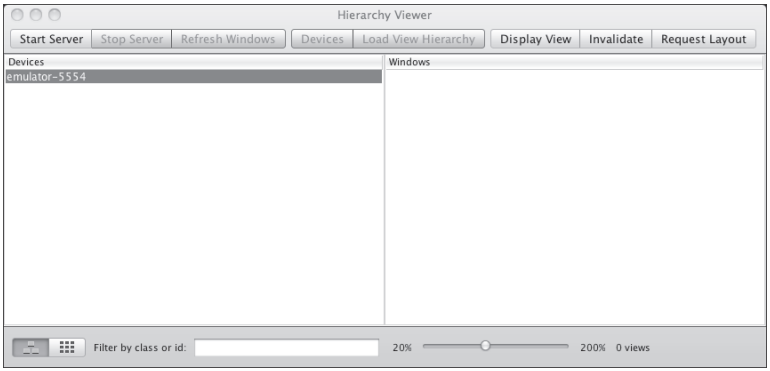


Figure 35.2
Fenêtre principale du visualisateur hiérarchique.

La liste à gauche montre les différents émulateurs que vous avez chargés. Le nombre après le tiret devrait correspondre au nombre entre parenthèses situé dans la barre de titre de l'émulateur.

Si vous cliquez sur le bouton Start Server et sur un émulateur, la liste des fenêtres accessibles apparaît à droite, comme le montre la Figure 35.3.

Vous remarquerez qu'outre l'activité ouverte apparaissent de nombreuses autres fenêtres, dont celle du lanceur (l'écran d'accueil), celle du "Keyguard" (l'écran noir Appuyez sur

Menu pour déverrouiller le téléphone qui apparaît lorsque vous ouvrez l'émulateur pour la première fois), etc. Votre activité est identifiée par le nom du paquetage et de la classe de l'application (`com.commonsware.android.readwrite/...`, ici).

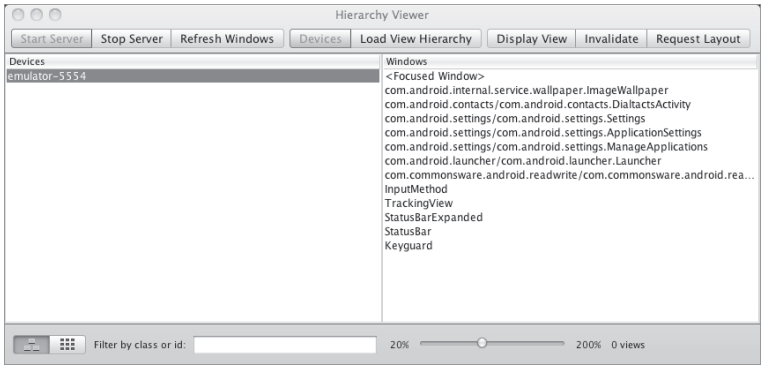


Figure 35.3
Hiérarchie des fenêtres disponibles.

Les choses commencent à devenir intéressantes lorsque vous sélectionnez l'une de ces fenêtres et que vous cliquez sur le bouton Load View Hierarchy. Après quelques secondes, les détails apparaissent dans une fenêtre appelée Layout View (voir Figure 35.4).

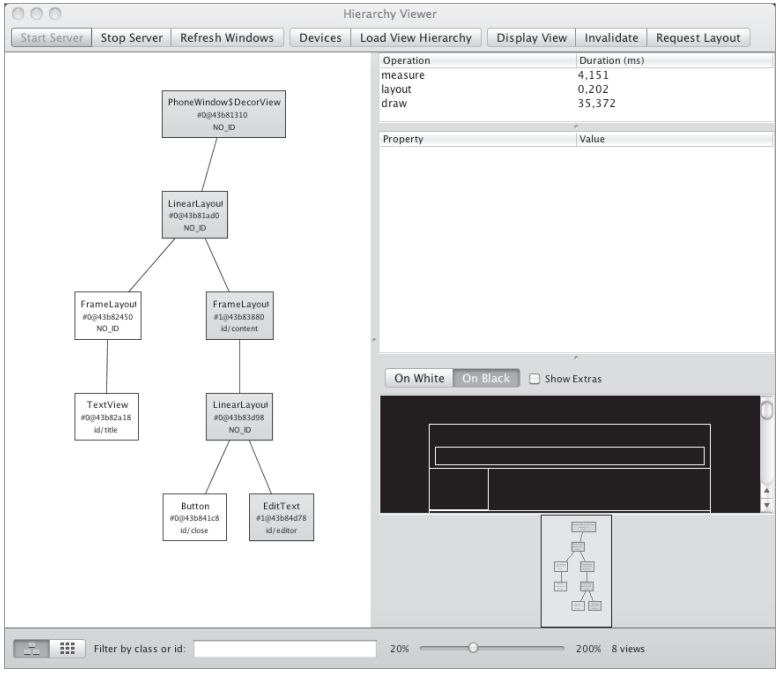


Figure 35.4
Layout View de l'application `ReadWriteFileDemo`.

La zone principale de cette Layout View est occupée par une arborescence des différentes vues qui composent votre activité, en partant de la fenêtre principale du système et en descendant vers les différents widgets graphiques que verra l'utilisateur. Dans la branche inférieure droite de cet arbre, vous retrouverez les widgets `LinearLayout`, `Button` et `EditText` utilisés par l'application. Les autres vues sont toutes fournies par le système, y compris la barre de titre.

Si vous cliquez sur l'une de ces vues, des informations supplémentaires apparaissent dans le visualisateur, comme le montre la Figure 35.5.

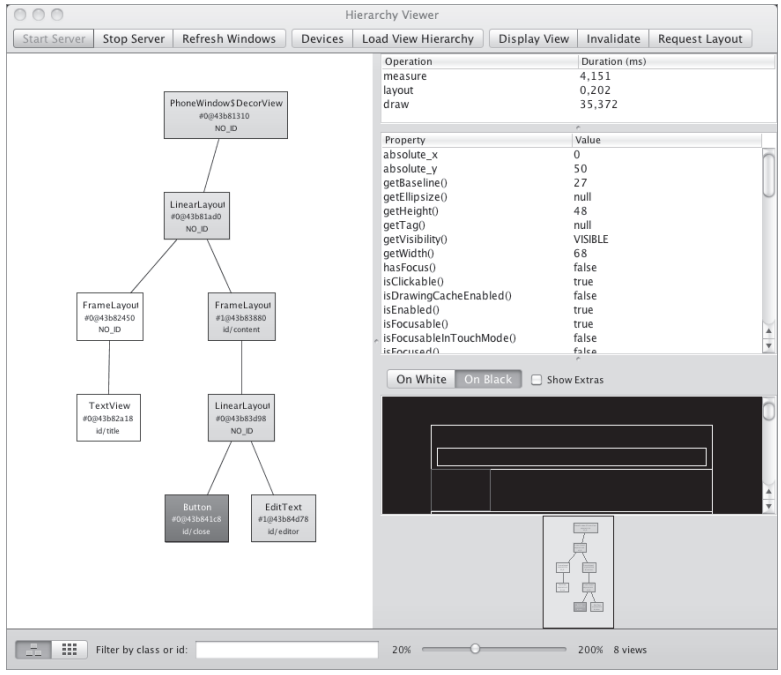


Figure 35.5
Affichage des propriétés d'une vue.

Dans la région supérieure droite du visualisateur, nous pouvons maintenant voir les propriétés du widget sélectionné – ici, le bouton. Malheureusement, ces propriétés ne sont pas modifiables.

En outre, le widget sélectionné est surligné en rouge dans la représentation schématique de l'activité qui apparaît sous la liste des propriétés (par défaut, les vues sont représentées par des contours blancs sur un fond noir) : ceci permet de vérifier que vous avez sélectionné le bon widget lorsque, par exemple, il y a plusieurs boutons et que la représentation de l'arborescence ne permet pas de les distinguer.

Si vous double-cliquez sur une vue de l'arborescence, un panneau apparaît pour ne vous montrer que cette vue (et ses fils), isolée du reste de l'activité.

Dans le coin inférieur gauche de la fenêtre principale se trouvent deux boutons – celui qui représente une arborescence est choisi par défaut. Si vous cliquez sur le bouton qui représente une grille, le visualisateur affiche une autre représentation, appelée "Pixel Perfect View" (voir Figure 35.6).

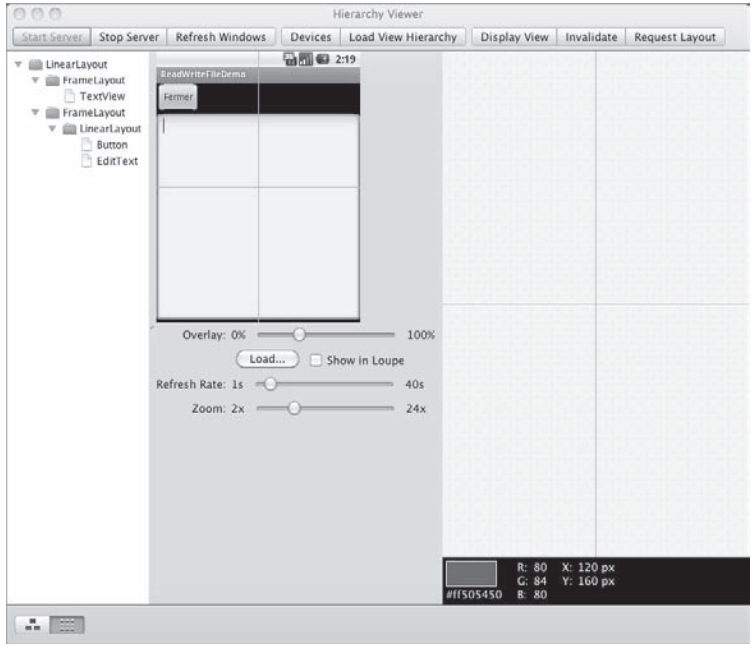


Figure 35.6
Visualisateur hiérarchique en mode "Pixel Perfect View".

La partie gauche contient une représentation des widgets et des autres vues de votre activité. Au milieu se trouve votre activité ("Normal View") et, sur la droite, vous pouvez voir une version zoomée ("Loupe View") de celle-ci.

Il faut bien comprendre que cette visualisation est en direct : l'activité est interrogée selon la fréquence choisie par le curseur "Refresh Rate". Tout ce que vous faites avec cette activité se reflétera donc dans les vues "Normal" et "Loupe" de la fenêtre "Pixel Perfect View".

Les lignes fines de couleur cyan placées au-dessus de l'activité montrent la position sur laquelle le zoom s'applique – il suffit de cliquer sur une nouvelle zone pour changer l'endroit inspecté par la "Loupe View". Un autre curseur permet de régler la puissance du grossissement.

DDMS (*Dalvik Debug Monitor Service*)

L'autre outil de l'arsenal du développeur Android s'appelle DDMS (*Dalvik Debug Monitor Service*). C'est une sorte de "couteau suisse" qui vous permet de parcourir les fichiers journaux, de modifier la position GPS fournie par l'émulateur, de simuler la réception d'appels et de SMS et de parcourir le contenu de l'émulateur pour y placer ou en extraire des fichiers. Nous ne présenterons ici que les fonctionnalités les plus utiles.

Pour utiliser DDMS, lancez le programme `ddms` qui se trouve dans le répertoire `tools/` de votre installation du SDK. Au départ, vous ne verrez dans la partie gauche qu'une arborescence des émulateurs avec les programmes qu'ils exécutent (voir Figure 35.7).

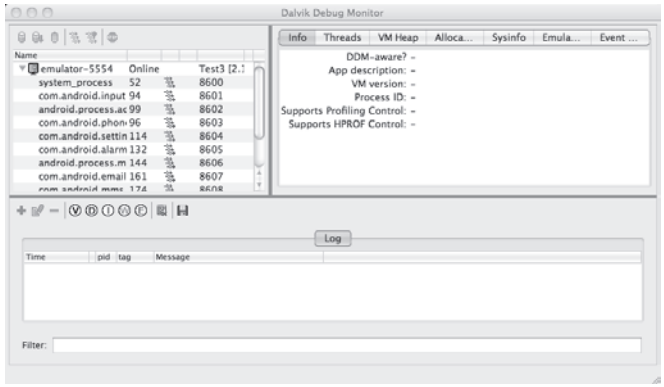


Figure 35.7
Vue initiale de DDMS.

Cliquer sur un émulateur permet de parcourir le journal des événements qui apparaît dans la zone du bas et de manipuler l'émulateur *via* un onglet qui se trouve à droite (voir Figure 35.8).

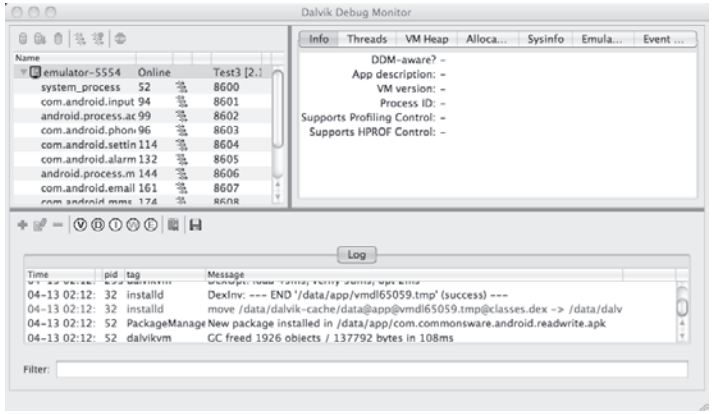


Figure 35.8
Émulateur sélectionné dans DDMS.

Journaux

À la différence d'`adb logcat`, DDMS vous permet d'examiner le contenu du journal dans un tableau doté d'une barre de défilement. Il suffit de cliquer sur l'émulateur ou le terminal que vous voulez surveiller pour que le bas de la fenêtre affiche le contenu du journal.

En outre, vous pouvez effectuer les opérations suivantes :

- Filtrer les entrées du journal selon l'un des cinq niveaux représentés par les boutons `V` à `E` dans la barre d'outils.
- Créer un filtre personnalisé pour ne voir que les entrées correspondantes. Pour ce faire, cliquez sur le bouton + et remplissez le formulaire : le nom que vous choisirez pour ce filtre sera utilisé pour nommer un autre onglet qui apparaîtra à côté du contenu du journal (voir Figure 35.9).
- Sauvegarder les entrées du journal dans un fichier texte, afin de pouvoir les réutiliser plus tard.

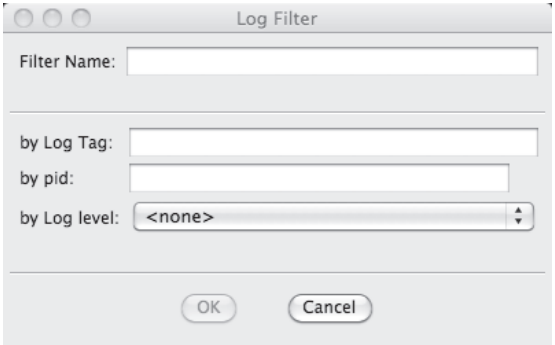
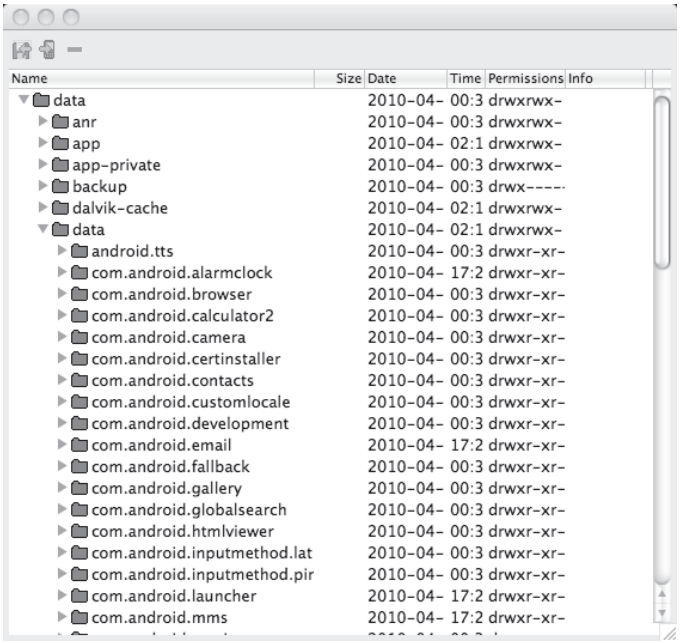


Figure 35.9
Filtrage des entrées du journal avec DDMS.

Stockage et extraction de fichiers

Bien que vous puissiez utiliser `adb pull` et `adb push` pour, respectivement, extraire ou stocker des fichiers sur un émulateur ou un terminal, DDMS permet de le faire de façon plus visuelle. Il suffit, pour cela, de sélectionner l'émulateur ou le terminal concerné, puis de choisir l'option Device > File Explorer... à partir du menu principal : une fenêtre de dialogue typique, comme celle de la Figure 35.10, permet alors de parcourir l'arborescence des fichiers.

Figure 35.10
Explorateur de fichiers de DDMS.



Sélectionnez simplement le fichier concerné et cliquez sur le bouton d'extraction (à gauche) ou de stockage (au milieu) de la barre d'outils pour le transférer vers ou à partir de votre machine de développement. Le bouton de suppression (à droite) permet de supprimer le fichier sélectionné.

- L'explorateur de fichiers a toutefois quelques limites :Il ne permet pas de créer de répertoire : pour cela, vous devez soit utiliser la commande `adb shell`, soit les créer à partir de votre application.
- Bien que vous puissiez parcourir la plupart des fichiers d'un émulateur, les restrictions de sécurité d'Android limitent beaucoup l'accès en dehors de l'arborescence `/sdcard` sur un vrai terminal.

Copies d'écran

Pour faire une copie d'écran de l'émulateur ou d'un terminal Android, cliquez sur Ctrl+S ou choisissez Device > Screen capture... dans le menu principal. Ceci ouvrira une boîte de dialogue contenant une image de l'écran courant, comme à la Figure 35.11.

Figure 35.11
Capture d'écran avec DDMS.

À partir de là, vous pouvez cliquer sur Save pour sauvegarder l'image au format PNG sur votre machine de développement, rafraîchir l'image à partir de l'état courant de l'émulateur ou du terminal, ou cliquer sur Done pour fermer la boîte de dialogue.

Mise à jour de la position

Pour que DDMS mette à jour la position de votre application, vous devez d'abord faire en sorte que l'application utilise le fournisseur de localisation `gps`, car c'est le seul que DDMS sait modifier. Puis cliquez sur l'onglet Emulator Control et recherchez la section Location Controls. Dans celle-ci, vous trouverez un cadre avec trois onglets permettant de choisir le format des coordonnées : Manual, GPX et KML (voir Figure 35.12).

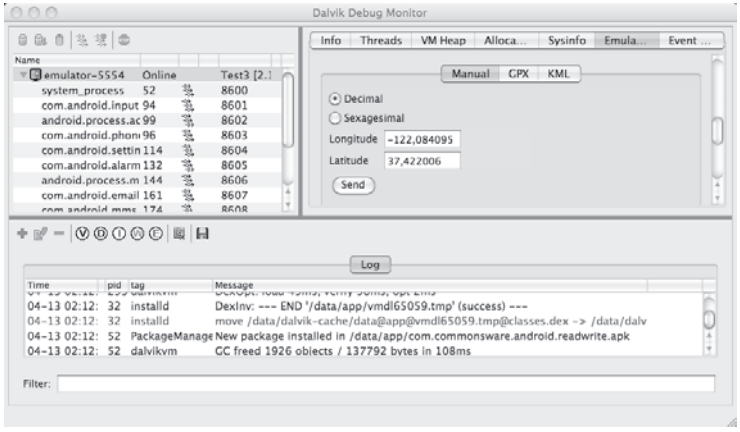


Figure 35.12
Contrôle de la position avec DDMS.

L'utilisation de l'onglet Manual est assez évidente : on fournit une latitude et une longitude et l'on clique sur le bouton Send pour envoyer cet emplacement à l'émulateur. Ce dernier, à son tour, préviendra les écouteurs de localisation de la nouvelle position.

Les autres onglets permettent de préciser des emplacements aux formats GPX (*GPS eXchange*) ou KML (*Keyhole Markup Language*).

Appels téléphoniques et SMS

DDMS sait également simuler la réception d'appels téléphoniques et de SMS *via* le groupe "Telephony Actions" de l'onglet Emulator Control (voir Figure 35.13).

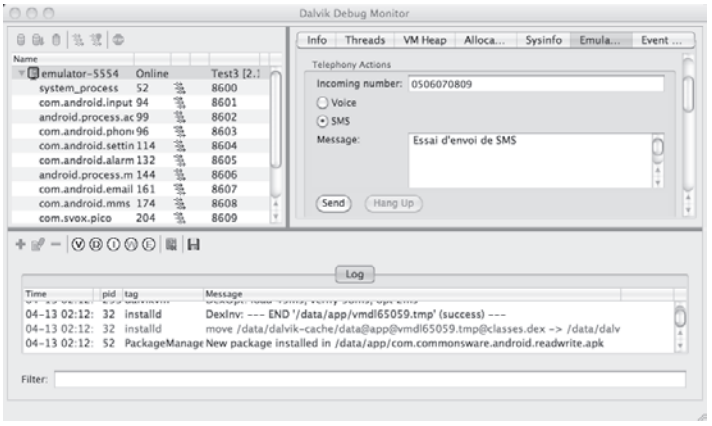


Figure 35.13
Contrôle de la téléphonie avec DDMS.

Pour simuler un appel téléphonique, saisissez un numéro, cochez le bouton radio Voice puis cliquez sur le bouton Call. Comme le montre la Figure 35.14, l'émulateur affichera l'appel entrant et vous demandera si vous l'acceptez (avec le bouton vert du téléphone) ou si vous le rejetez (avec le bouton rouge).

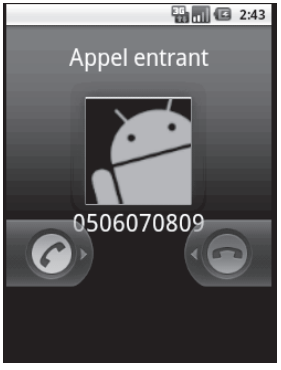


Figure 35.14
Simulation de la réception d'un appel.

Pour simuler la réception d'un SMS, saisissez un numéro téléphonique, cochez le bouton radio SMS, saisissez un message dans la zone de texte et cliquez sur Send. Le message apparaîtra sous la forme d'une notification, comme à la Figure 35.15.

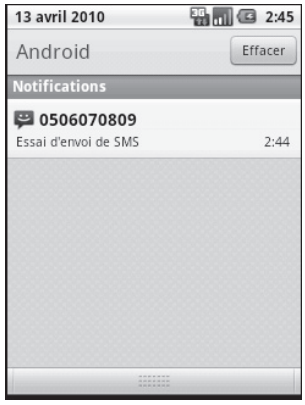


Figure 35.15
Simulation de la réception d'un SMS.

En cliquant sur la notification, vous pourrez voir le contenu intégral du message, comme le montre la Figure 35.16.

Figure 35.16
Simulation de la réception d'un SMS dans l'application SMS/MMS.

Gestion des cartes amovibles

Le G1 dispose d'un emplacement pour carte microSD et de nombreux autres terminaux Android disposent d'une forme similaire de stockage amovible, désigné de façon générique par le terme "Carte SD".

Les développeurs sont fortement encouragés à utiliser les cartes SD pour y stocker les gros fichiers, comme les images, les clips vidéo, les fichiers musicaux, etc. La mémoire interne du G1, notamment, est relativement peu importante et il est donc préférable de stocker un maximum de données sur une carte SD.

Bien que le G1 ait une carte SD par défaut, le problème, évidemment, est que l'émulateur n'en a pas. Pour que ce dernier se comporte comme le G1, vous devez donc créer et "insérer" une carte SD dans l'émulateur.

Création d'une image de carte

Au lieu d'exiger que les émulateurs aient accès à un vrai lecteur de carte SD pour utiliser de vraies cartes, Android est configuré pour utiliser des images de cartes. Une image est simplement un fichier que l'émulateur traitera comme s'il s'agissait d'un volume de carte SD : il s'agit en fait du même concept que celui utilisé par les outils de virtualisation (comme Virtual Box) – Android utilise une image disque pour représenter le contenu d'une carte SD.

Pour créer cette image, utilisez le programme `mksdcard` qui se trouve dans le répertoire `tools/` de votre installation du SDK. Ce programme attend au moins deux paramètres :

- La taille de l'image et donc de la "carte". Si vous fournissez simplement un nombre, celui-ci sera interprété comme un nombre d'octets. Vous pouvez également le faire suivre de `K` ou `M` pour préciser que cette taille est, respectivement, exprimée en kilo-octets ou en mégaoctets.
- Le nom du fichier dans lequel stocker l'image.

Pour, par exemple, créer l'image d'une carte SD de 1 Go afin de simuler celle du GI dans l'émulateur, utilisez la commande suivante:

```
mksdcard 1024M sdcard.img
```

```
Insertion de la carte
```

Pour que l'émulateur utilise cette image de carte, lancez-le avec l'option `-sdcard` suivie du chemin complet vers le fichier image créé avec `mksdcard`. Bien que cette option n'ait pas d'effet visible – aucune icône d'Android ne montrera qu'une carte est montée –, le répertoire `/sdcard` sera désormais accessible en lecture et en écriture.

Pour placer et lire des fichiers dans `/sdcard`, il suffit ensuite d'utiliser l'explorateur de fichiers de DDMS ou les commandes `adb push` et `adb pull` à partir de la console.

Les manipulations de la SD Card (création et insertion) seront simplifiées par l'utilisation d'ADT en complétant le champs de saisie nommé SD Card Size lors de la création de l'AVD.

36

Gestion des différentes tailles d'écran

Les écrans des terminaux Android, qui sont apparus sur le marché l'année qui a suivi l'apparition d'Android 1.0, avaient tous la même résolution (HVGA, 320 × 480) et la même taille (environ 3,5 pouces, soit 9 cm). À partir de l'automne 2009, cependant, des terminaux disposant d'écrans de tailles et de résolutions très différentes ont vu le jour, du minuscule QVGA (240 × 320) au WVGA (480 × 800), bien plus grand.

Les utilisateurs s'attendent bien sûr à ce que les applications fonctionnent sur tous ces types d'écrans et tirent éventuellement parti des écrans plus larges. Android 1.6 a donc ajouté un meilleur support permettant aux développeurs de mieux gérer ces différentes tailles et résolutions.

La documentation d'Android explique en détail les techniques de gestion des écrans (**http://d.android.com/guide/practices/screens_support.html**) : nous vous conseillons de la lire en même temps que ce chapitre pour comprendre au mieux ce que vous devrez gérer et pour tirer le meilleur parti possible des différentes tailles d'écrans.

Après un certain nombre de sections présentant les options et la théorie, ce chapitre présentera une application simple mais sachant s'adapter aux différents écrans existants.

Écran par défaut

Commençons par ignorer totalement cette problématique des tailles et des résolutions d'écrans.

Si votre application a été compilée pour Android 1.5 ou une version antérieure, Android supposera qu'elle a été conçue pour s'afficher correctement avec une taille et une résolution d'écran classiques. Si vous l'installez sur un terminal doté d'un écran plus grand, Android l'exécutera automatiquement en *mode compatible*, consistant à adapter son échelle à la taille réelle de l'écran du terminal.

Supposons, par exemple, que vous utilisiez un fichier PNG de 24 × 24 pixels et que votre application s'exécute sur un terminal ayant un écran de taille normale, mais avec une résolution WVGA (c'est ce que l'on appelle un écran haute densité). Android modifiera la taille du fichier PNG pour qu'il fasse 36 × 36 pixels afin d'occuper toute la surface visible de l'écran. L'avantage est que ce traitement est automatique ; l'inconvénient est que les algorithmes de changement d'échelle ont tendance à produire des images moins nettes.

En outre, Android empêchera votre application de s'exécuter sur des terminaux ayant des écrans plus petits – les terminaux QVGA comme le HTC Tattoo, par exemple – bien qu'elle soit disponible sur l'Android Market.

Si l'application a été compilée pour Android 1.6 ou une version supérieure, par contre, Android supposera qu'elle sait correctement gérer toutes les tailles d'écran et ne l'exécutera donc pas en mode compatible. Nous verrons plus loin comment configurer tout cela.

Tout en un

L'approche la plus simple pour gérer les tailles d'écrans multiples avec Android consiste à concevoir l'interface utilisateur pour qu'elle s'adapte automatiquement à la taille de l'écran, sans indiquer de taille précise dans le code Java ou les ressources – en d'autres termes, faire en sorte que "ça marche".

Ceci implique, toutefois, que tout ce qu'utilise l'interface utilisateur soit bien redimensionné par Android et que tout se place correctement, même sur un écran QVGA.

Les sections qui suivent donnent quelques conseils pour obtenir ce résultat.

Penser en termes de règles, pas en termes de positions

Certains développeurs, notamment ceux qui ont été formés à l'école du développement des interfaces (drag and drop notamment), pensent d'abord et avant tout à la position des widgets. Ils veulent des widgets précis avec des tailles fixées et à des positions bien déterminées. Les gestionnaires de placement (conteneurs) d'Android les ennuient et ils utilisent donc parfois le conteneur obsolète `AbsoluteLayout` pour concevoir leurs interfaces comme ils en avaient l'habitude.

Cette approche fonctionne rarement bien, même sur les ordinateurs de bureau, comme on peut le constater avec certaines applications qui ont du mal à gérer le changement de taille des fenêtres. De la même façon, elle ne fonctionne pas sur les terminaux mobiles, qui proposent désormais un large éventail de tailles et de résolutions d'écran.

Au lieu de penser en termes de positions, vous devez réfléchir en termes de *règles*. Vous devez expliquer à Android les règles métier concernant les tailles et les positions de vos widgets et le laisser interpréter ces règles en fonction de la résolution de l'écran du terminal sur laquelle votre application est déployée.

Les règles les plus simples consistent à utiliser les valeurs `fill_parent` et `wrap_content` pour les attributs `android:layout_width` et `android:layout_height` car elles ne précisent aucune taille spécifique et s'adaptent à l'espace disponible.

`RelativeLayout` (voir Chapitre 6) est l'environnement qui permet de tirer le meilleur parti de ces règles. Bien qu'il semble compliqué au premier abord, `RelativeLayout` permet de contrôler le positionnement des widgets tout en l'adaptant aux différentes tailles d'écran. Vous pouvez donc, par exemple :

- Positionner explicitement les widgets en bas ou à droite de l'écran plutôt qu'espérer qu'ils se placeront là par le bon vouloir d'un autre layout.
- Contrôler les distances entre les widgets associés (le label d'un champ devrait être à sa gauche, par exemple), sans jouer sur le remplissage ou les marges.

Pour avoir le maximum de contrôle sur les règles, l'idéal serait de créer votre propre classe de layout. Si, par exemple, vous développez une suite d'applications de jeux de cartes, il serait préférable de disposer d'une classe spécialisée dans le placement des cartes – la façon qu'elles ont de se recouvrir, le placement des cartes retournées ou non, la façon de prendre plusieurs cartes à la fois, etc. Même si vous pourriez obtenir l'aspect recherché avec un `RelativeLayout`, par exemple, vous auriez intérêt à implémenter une classe `PlayingCardLayout` spécialement conçue pour ce genre d'applications. Malheureusement, il n'existe pas toujours de documentation concernant la création de ces layouts personnalisés.

Utilisez des dimensions physiques

Android permet d'exprimer les dimensions dans un grand nombre d'unités. La plus utilisée était le pixel (`px`) car on se le représente facilement mentalement : les dimensions des écrans Android sont d'ailleurs exprimées en nombre de pixels en largeur et en hauteur.

Cependant, les pixels commencent à poser problème lorsque la densité des écrans change. En effet, la taille des pixels diminue à mesure que le nombre de pixels augmente pour une taille d'écran donnée : une icône de 32 pixels sur un terminal Android classique peut convenir aux touchés alors que, sur un écran à haute densité (un téléphone mobile WVGA, par exemple), elle peut être trop petite pour être manipulable au doigt.

Si vous avez exprimé en pixels la taille d'un widget comme un bouton, utilisez plutôt des millimètres (`mm`) ou des pouces (`in`) comme unité de mesure : 10 millimètres feront toujours 10 millimètres, quelle que soit la résolution et la taille de l'écran. Vous pouvez ainsi garantir que ce widget conviendra toujours à une manipulation tactile, quel que soit le nombre de pixels qu'il occupera.

Évitez les vrais pixels

Dans les cas où exprimer une dimension en millimètres n'aurait aucun sens, vous pouvez quand même utiliser d'autres unités de mesure et éviter les vrais pixels.

Android permet d'exprimer les dimensions en pixels indépendants de la densité (`dip`, pour *density-independent pixel*). Ces pixels correspondent exactement aux vrais pixels sur un écran de 160 dpi (celui d'un terminal HVGA classique, par exemple) et s'adaptent ensuite en conséquence. Sur un écran de 240 dpi (celui d'un téléphone WVGA, par exemple), un `dip` correspond à 1,5 `px` : 50 `dip` correspondent donc à 50 `px` à 160 dpi et à 75 `px` à 240 dpi. L'avantage des `dip` est donc que la taille réelle de la dimension reste la même et qu'il n'y a aucune différence visuelle entre 50 `dip` à 160 dpi et 50 `dip` à 250 dpi.

Android autorise également les *scaled pixels* (`sp`), dont l'échelle dépend, en théorie, de la taille de la police choisie (la valeur `FONT_SCALE` dans `System.Settings`).

Choisir des images adaptables

La taille des images bitmaps – PNG, JPG et GIF – ne s'adapte pas naturellement. Si vous n'utilisez pas le mode compatible, Android n'essaiera donc même pas de les adapter à la résolution et à la taille de l'écran. La taille initiale de votre image sera celle utilisée pour son affichage, même si elle est trop petite ou trop grande pour certains écrans.

Un moyen de régler ce problème consiste à éviter les images bitmaps statiques et à les remplacer par des bitmaps *nine-patch* et des images définies en XML (avec `GradientDrawable`, par exemple). Une image bitmap nine-patch est un fichier PNG encodé avec des

règles indiquant comment l'image peut s'agrandir pour occuper plus d'espace. Les images définies en XML utilisent, quant à elles, un langage ressemblant à SVG pour définir leurs formes, le tracé de leurs traits, leurs motifs de remplissage, etc.

Fait maison, rien que pour vous...

Lorsque vous avez besoin de comportements et d'aspects différents en fonction des tailles ou des densités d'écran, Android vous permet de choisir des ressources ou des blocs de code selon l'environnement dans lequel s'exécute votre application. Utilisé avec les techniques présentées dans la section précédente, ceci permet d'obtenir une indépendance vis-à-vis de la taille et de la densité des écrans, en tout cas pour les terminaux qui utilisent Android 1.6 ou une version plus récente.

<supports-screens>

La première étape pour gérer activement les tailles d'écrans consiste à ajouter un élément `<supports-screens>` à votre fichier `AndroidManifest.xml`. Cet élément permet d'indiquer explicitement les tailles d'écran que vous gérez et celles que vous ne gérez pas. Celles qui ne sont pas explicitement gérées seront prises en charge par le mode compatible automatique que nous avons décrit plus haut.

Voici un manifeste contenant un élément `<supports-screens>` :

```
<?xml version="1.0" encoding="utf-8"?>
  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
            package="com.commonsware.android.eu4you"
            android:versionCode="1"
            android:versionName="1.0">
    <supports-screens
            android:largeScreens="true"
            android:normalScreens="true"
            android:smallScreens="true"
            android:anyDensity="true"
    />
    <application android:label="@string/app_name"
                 android:icon="@drawable/cw">
      <activity android:name=".EU4You"
                android:label="@string/app_name">
        <intent-filter>
          <action android:name="android.intent.action.MAIN" />
```

```
            <category android:name="android.intent.category.LAUNCHER" />
          </intent-filter>
        </activity>
      </application>
    </manifest>
```

Les attributs `android:smallScreens`, `android:normalScreens` et `android:largeScreens` prennent une valeur booléenne indiquant si votre application gère explicitement ces écrans (`true`) ou nécessite l'assistance du mode compatible (`false`).

L'attribut `android:anyDensity` précise que vous tenez compte de la densité dans vos calculs (`true`) ou non (`false`). Dans ce dernier cas, Android se comportera comme si toutes vos dimensions (`4px`, par exemple) étaient relatives à un écran de densité normale (160 dpi). Si l'application s'exécute sur un terminal avec une densité plus faible ou plus élevée, Android adaptera en conséquence vos dimensions. Si cet attribut vaut `true`, Android vous laissera la responsabilité d'utiliser des unités indépendantes de la densité, comme `dip`, `mm` ou `in`.

Ressources et ensembles de ressources

Le principal moyen de choisir les ressources en fonction de la taille ou de la densité de l'écran consiste à créer des ensembles de ressources spécifiques aux différentes caractéristiques d'écran. Vous expliquez ainsi à Android comment gérer chacune de ces ressources et il choisira automatiquement l'ensemble qui convient.

Taille par défaut

Par défaut, Android adaptera toutes les ressources *drawable*. Celles qui peuvent l'être naturellement seront donc correctement redimensionnées. En revanche, la taille des images bitmaps traditionnelles sera recalculée à l'aide d'un algorithme classique, ce qui peut, en fonction des situations, donner ou non de bons résultats ; par ailleurs, ce calcul peut également ralentir un peu l'application. Pour éviter cela, mettez en place des ensembles de ressources distincts contenant des images bitmaps non adaptables.

Ensembles en fonction de la densité

Si vous souhaitez disposer de layouts ou de dimensions différentes – ou tout ce qui dépend de la densité de l'écran –, utilisez les labels `-ldpi`, `-mdpi` et `-hdpi` pour les ensembles de ressources. Le fichier `res/values-hdpi/dimens.xml`, par exemple, contiendra les dimensions utilisées pour les écrans de haute densité.

Ensembles en fonction de la taille

De même, pour les ensembles de ressources dépendant de la taille de l'écran, Android utilise les labels `-small`, `-normal` et `-large`. Le répertoire `res/layout-large-land/` contiendra donc les layouts pour les grands écrans (WVGA, par exemple) en mode paysage.

Ensembles en fonction de la version

Les anciennes versions d'Android peuvent parfois être perturbées par ces nouveaux labels d'ensembles de ressources. Vous pouvez alors ajouter un label de version sous la forme `-v`*N*, où *N* est le niveau de l'API. Ainsi, `res/drawable-large-v4/` indique que ces drawables seront utilisés sur de grands écrans pour l'API de niveau 4 (Android 1.6) ou une version plus récente.

Android sachant filtrer les versions depuis un certain temps, cette technique fonctionne depuis Android 1.5 (voire éventuellement avec les versions antérieures). Par conséquent, si vous constatez que les émulateurs ou les terminaux Android 1.5 choisissent le mauvais ensemble de ressources, ajoutez `-v4` aux noms de ces ensembles afin de les filtrer.

Trouver sa taille

Voici quelques possibilités permettant à votre code Java d'agir différemment en fonction de la taille ou de la densité de l'écran du terminal sur lequel il s'exécute.

Si vos ensembles de ressources sont différentiables, exploitez cette différence pour les utiliser de façon appropriée dans votre code. Comme nous le verrons plus loin dans ce chapitre, certains layouts peuvent en effet avoir des widgets supplémentaires (`res/layout-large/main.xml`, par exemple) : il suffit alors de vérifier que l'un de ces widgets est présent pour savoir que vous vous exécutez sur un écran large.

Vous pouvez également connaître la classe de votre taille d'écran en passant par un objet `Configuration` que vous obtiendrez généralement par un appel à `getResources().getConfiguration()` sur l'activité. Cet objet contient un champ public `screenLayout` qui est un masque indiquant le type d'écran sur lequel s'exécute l'application : vous pouvez donc tester si l'écran est petit, normal ou grand, ou s'il est long (c'est-à-dire au format 16:9 et non 4:3). Voici, par exemple, comment tester que l'on s'exécute sur un grand écran :

```
if (getResources().getConfiguration().screenLayout
    & Configuration.SCREENLAYOUT_SIZE_LARGE) ==
    Configuration.SCREENLAYOUT_SIZE_LARGE) {
  // C'est un grand écran
} else {
  // Ce n'est pas un grand écran
}
```

En revanche, on ne peut pas obtenir la densité de l'écran avec une technique équivalente. Si vous devez absolument la connaître, une astuce consiste à créer des répertoires `res/values-ldpi/`, `res/values-mdpi/` et `res/values-hdpi/` dans votre projet en leur ajoutant un fichier `strings.xml` à chacun. Placez ensuite une ressource textuelle de même nom dans chaque `strings.xml`, mais avec une valeur différente (appelez-la par exemple `density` et donnez-lui, respectivement, les valeurs `ldpi`, `mdpi` et `hdpi`). Vous pouvez ensuite tester la valeur de cette ressource lors de l'exécution. Ce n'est pas très élégant, mais cela devrait fonctionner.

Rien ne vaut la réalité

Les émulateurs Android vous aideront à tester votre application sur des écrans de différentes tailles. Cependant, ceci ne vous emmènera pas bien loin car les écrans LCD des terminaux portables n'ont pas les mêmes caractéristiques que les écrans des ordinateurs portables ou de bureau :

- Les écrans LCD des mobiles ont une densité bien supérieure à l'écran de votre machine de développement.
- Une souris permet d'effectuer un touché écran bien plus précis que ne le permet un doigt.

Lorsque cela est possible, vous devez donc soit utiliser l'émulateur de façon originale, soit utiliser un véritable terminal avec d'autres résolutions d'écrans.

Différences de densité

Le DROID de Motorola a un écran de 240 dpi, de 3,7 pouces et de 480 × 854 pixels (affichage FWVGA). Pour l'émuler en tenant compte du nombre de pixels, vous devez utiliser un tiers d'un moniteur LCD de 19 pouces et de 1 280 × 1 024 pixels car la densité du moniteur est bien plus faible que celle de l'écran du DROID – environ 96 dpi. Lorsque vous lancerez l'émulateur Android pour un affichage FWVGA comme celui du DROID, vous obtiendrez donc une fenêtre d'émulation énorme.

Ceci est tout à fait valable pour déterminer l'aspect général de votre application dans un environnement FWVGA. Quelle que soit la densité, les widgets s'aligneront de la même façon, les tailles auront les mêmes rapports (un widget A sera deux fois plus grand qu'un widget B, quelle que soit la densité, par exemple), etc.

Cependant, les problèmes suivants peuvent se poser :

- Ce qui semblait être d'une taille correcte sur un écran de 19 pouces peut se révéler vraiment trop petit sur l'écran d'un mobile avec la même résolution.

- Ce que vous pouviez aisément cliquer avec une souris dans l'émulateur peut être impossible à sélectionner avec le doigt sur un écran plus petit et plus dense.

Ajustement de la densité

Par défaut, l'émulateur conserve le nombre précis de pixels au détriment de la densité – c'est la raison pour laquelle sa fenêtre est si grande – mais vous pouvez également choisir de conserver la densité au détriment du nombre de pixels.

Pour cela, le moyen le plus simple consiste à utiliser le gestionnaire d'AVD introduit par Android 1.6. Comme le montre la Figure 36.1, la version de cet outil fournie avec le SDK d'Android 2.0 dispose d'une boîte de dialogue Launch Options qui s'ouvre lorsque vous lancez une instance de l'émulateur en cliquant sur le bouton Start.

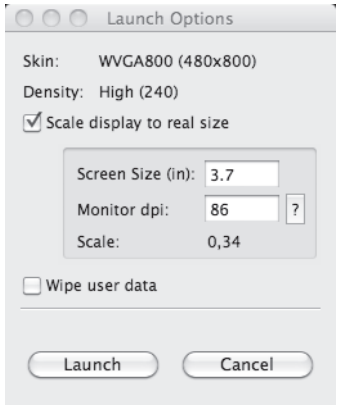


Figure 36.1
La boîte de dialogue Launch Options.

Par défaut, la case Scale display to real size n'est pas cochée et Android ouvrira la fenêtre de l'émulateur normalement, mais vous pouvez la cocher afin de fournir deux informations sur la taille de l'écran :

- La taille de l'écran du terminal que vous souhaitez émuler, en pouces (3,7 pour le DROID de Motorola, par exemple).
- La résolution en points par pouces (dpi) de votre moniteur (cliquez sur le bouton ? pour afficher une calculatrice permettant de déterminer cette valeur).

Vous obtiendrez ainsi une fenêtre d'émulation qui représente plus fidèlement l'aspect qu'aura votre interface graphique sur un terminal physique, au moins en termes de taille. Cependant, l'émulateur utilisant bien moins de pixels qu'un vrai terminal, les polices seront peut-être difficiles à lire, les images seront pixellisées, etc.

Accès aux véritables terminaux

Le meilleur moyen possible de voir à quoi ressemblera votre application sur différents terminaux consiste, évidemment, à la tester sur ces terminaux. Vous n'avez pas nécessairement besoin de disposer de tous les modèles Android existants, mais uniquement de ceux ayant des caractéristiques matérielles distinctives, susceptibles d'avoir un impact sur l'application – et la taille des écrans influe à peu près sur tout. Voici quelques suggestions :

- Testez virtuellement les terminaux en utilisant des services comme DeviceAnywhere (**http://www.deviceanywhere.com/**). Bien que ce soit une version améliorée de l'émulateur, elle n'est pas gratuite et ne permet pas de tout tester non plus (les changements d'emplacements, par exemple).
- Vous pouvez acheter des terminaux d'occasion sur Ebay, par exemple. Les GSM "désimlockés" peuvent partager une carte SIM lorsque vous voulez tester des opérations de téléphonie ou fonctionner sans carte SIM.
- Si vous vivez en ville, vous pouvez créer un groupe d'utilisateurs et l'utiliser pour tester les applications sur les différents terminaux des membres de ce groupe.
- Servez-vous des retours des utilisateurs : produisez une version bêta gratuite de votre application et laissez les impressions des utilisateurs guider vos modifications. Vous pouvez distribuer ces versions en dehors de l'Android Market pour éviter que de mauvais retours sur les versions bêta fassent chuter le niveau d'appréciation de votre application.

Exploitez sans vergogne la situation

Pour l'instant, nous nous sommes attachés à faire en sorte que les layouts apparaissent correctement sur d'autres tailles d'écran. Pour les écrans plus petits que la moyenne (QVGA, par exemple), c'est probablement le mieux que l'on puisse faire.

Dès que l'on passe à des écrans plus grands, toutefois, une autre possibilité apparaît : l'utilisation de layouts différents pour tirer parti de cet espace supplémentaire. Ceci est tout particulièrement utile lorsque la taille physique de l'écran est plus grande (un LCD de 5 pouces comme celui de la tablette Android Archos 5, par exemple) car on préférera alors modifier la structure du layout pour profiter de cet espace.

Les sections qui suivent décrivent plusieurs moyens de profiter de cet espace supplémentaire.

Remplacer les menus par des boutons

Le choix d'un élément dans un menu d'options nécessite deux actions : presser le bouton du menu, puis toucher l'option choisie. Le choix d'un élément dans un menu contextuel

demande aussi deux actions : un touché prolongé sur le widget, suivi d'un touché de l'option choisie. En outre, les menus contextuels ont un problème supplémentaire dû au fait qu'ils sont invisibles : les utilisateurs, par exemple, peuvent ne pas se rendre compte que votre `ListView` dispose d'un menu contextuel.

Vous pourriez améliorer votre interface en fournissant un accès direct aux opérations qui sont ordinairement cachées dans un menu : non seulement cela diminue le nombre d'étapes nécessaires pour effectuer les actions, mais cela rend également ces actions plus visibles.

Supposons, par exemple, que vous développiez un lecteur musical et que vous souhaitiez offrir la possibilité de gérer manuellement des listes de morceaux. Une activité affiche les morceaux d'une liste dans une `ListView` et un menu d'options dispose d'un choix Ajouter permettant d'ajouter un nouveau morceau à la liste. Le menu contextuel de la liste dispose des options Supprimer, Déplacer vers le haut et Déplacer vers le bas pour réordonner les morceaux dans la liste. Sur un grand écran, vous pourriez ajouter quatre widgets `ImageButton` pour ces quatre options, les trois du menu contextuel n'étant actifs que lorsqu'une ligne a été sélectionnée au moyen du pad ou du trackball. Sur des écrans plus petits, vous vous contenteriez des menus.

Remplacer les onglets par une seule activité

Votre interface utilise peut-être un `TabHost` pour afficher plus de widgets dans l'espace disponible à l'écran. Tant que l'espace que vous économisez en plaçant les widgets dans un onglet séparé est supérieur à celui occupé par les onglets eux-mêmes, vous êtes gagnant. Cependant, utiliser plusieurs onglets implique également plus d'actions de l'utilisateur pour naviguer dans votre interface, notamment s'il doit passer régulièrement d'un onglet à l'autre.

Si vous n'utilisez que deux onglets, vous pouvez offrir plus d'espace à votre interface en les supprimant et en plaçant tous les widgets sur un seul écran : tout sera sous les yeux de l'utilisateur, qui n'aura plus besoin de naviguer dans les onglets.

Si vous avez plus de deux onglets, vous n'aurez sûrement pas assez d'espace pour placer tout leur contenu dans la même activité. Cependant, vous pouvez couper la poire en deux : faire en sorte que les widgets les plus utilisés soient toujours dans l'activité et laisser le `TabHost` gérer les autres sur l'autre moitié de l'écran.

Consolider les activités multiples

La technique la plus puissante consiste à profiter d'un écran plus grand pour se débarrasser des transitions entre les activités. Si, par exemple, vous utilisez une `ListActivity` dans laquelle un clic sur un élément place les détails de celui-ci dans une activité séparée, utilisez un layout où les détails seront dans la même activité que la `ListView` (la `ListView` sur

la gauche et les détails sur la droite dans un layout en mode paysage, par exemple). Ceci évitera à l'utilisateur de devoir sans cesse presser le bouton de retour en arrière pour quitter un ensemble de détails afin d'en voir un autre.

Nous utiliserons cette technique dans l'exemple présenté dans la section suivante.

Exemple : *EU4You*

Pour illustrer l'utilisation de quelques-unes des techniques que nous venons de présenter, nous examinerons l'application `ScreenSizes/EU4You`, qui n'a qu'une seule activité (`EU4You`) contenant une `ListView` proposant une liste des membres de l'Union européenne (UE) avec leurs drapeaux nationaux (**http://www.wpclipart.com/flags/Countries/index.html**). Un clic sur un pays affiche la page Wikipedia qui lui est consacrée.

Nous présenterons quatre versions de cette application en commençant par celle qui ne tient pas compte de la taille de l'écran et, petit à petit, nous lui ajouterons des fonctionnalités permettant de mieux l'adapter aux écrans des terminaux sur lesquels elle s'exécute.

Première version

Voici le fichier `AndroidManifest.xml`, qui ressemble à celui que nous avons présenté plus haut dans ce chapitre :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.commonsware.android.eu4you"
          android:versionName="1.0">
  <supports-screens android:largeScreens="true"
                    android:normalScreens="true"
                    android:smallScreens="true"
                    android:anyDensity="true"
  />
  <application android:label="@string/app_name" android:icon="@drawable/cw">
    <activity android:name=".EU4You"
              android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Notez que nous utilisons l'élément `<supports-screens>` pour indiquer que nous supportons toutes les tailles d'écran. Ceci permet d'empêcher le changement de taille automatique qu'Android effectuerait si nous annoncions que nous ne supportons pas certaines tailles.

Le fichier layout principal ne tient pas compte de la taille de l'écran car il définit simplement une `ListView` plein écran :

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

Une ligne, par contre, nécessite un peu de configuration :

```
<?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:padding="2dip"
                    android:minHeight="?android:attr/listPreferredItemHeight">
  <ImageView android:id="@+id/flag"
             android:layout_width="wrap_content"
             android:layout_height="wrap_content"
             android:layout_gravity="center_vertical|left"
             android:paddingRight="4px"
  />
  <TextView android:id="@+id/name"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center_vertical|right"
            android:textSize="20px"
  />
</LinearLayout>
```

Pour l'instant, nous utilisons donc une police de `20px`, qui ne variera pas en fonction de la taille ou de la densité de l'écran.

Notre activité `EU4You` est un peu longue, mais c'est essentiellement parce qu'il y a beaucoup de pays dans l'UE et que nous voulons afficher le drapeau et le texte sur une ligne :

```
package com.commonsware.android.eu4you;

import android.app.ListActivity;
import android.content.Intent;
```

```
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.ImageView;
import android.widget.ListView;
import android.widget.TextView;
import java.util.ArrayList;

public class EU4You extends ListActivity {
  static private ArrayList<Country> EU=new ArrayList<Country>();

  static {
    EU.add(new Country(R.string.austria, R.drawable.austria,
                       R.string.austria_url));
    EU.add(new Country(R.string.belgium, R.drawable.belgium,
                       R.string.belgium_url));
    EU.add(new Country(R.string.bulgaria, R.drawable.bulgaria,
                       R.string.bulgaria_url));
    EU.add(new Country(R.string.cyprus, R.drawable.cyprus,
                       R.string.cyprus_url));
    EU.add(new Country(R.string.czech_republic,
                       R.drawable.czech_republic,
                       R.string.czech_republic_url));
    EU.add(new Country(R.string.denmark, R.drawable.denmark,
                       R.string.denmark_url));
    EU.add(new Country(R.string.estonia, R.drawable.estonia,
                       R.string.estonia_url));
    EU.add(new Country(R.string.finland, R.drawable.finland,
                       R.string.finland_url));
    EU.add(new Country(R.string.france, R.drawable.france,
                       R.string.france_url));
    EU.add(new Country(R.string.germany, R.drawable.germany,
                       R.string.germany_url));
    EU.add(new Country(R.string.greece, R.drawable.greece,
                       R.string.greece_url));
    EU.add(new Country(R.string.hungary, R.drawable.hungary,
                       R.string.hungary_url));
```

```
    EU.add(new Country(R.string.ireland, R.drawable.ireland,
                       R.string.ireland_url));
    EU.add(new Country(R.string.italy, R.drawable.italy,
                       R.string.italy_url));
    EU.add(new Country(R.string.latvia, R.drawable.latvia,
                       R.string.latvia_url));
    EU.add(new Country(R.string.lithuania, R.drawable.lithuania,
                       R.string.lithuania_url));
    EU.add(new Country(R.string.luxembourg, R.drawable.luxembourg,
                       R.string.luxembourg_url));
    EU.add(new Country(R.string.malta, R.drawable.malta,
                       R.string.malta_url));
    EU.add(new Country(R.string.netherlands, R.drawable.netherlands,
                       R.string.netherlands_url));
    EU.add(new Country(R.string.poland, R.drawable.poland,
                       R.string.poland_url));
    EU.add(new Country(R.string.portugal, R.drawable.portugal,
                       R.string.portugal_url));
    EU.add(new Country(R.string.romania, R.drawable.romania,
                       R.string.romania_url));
    EU.add(new Country(R.string.slovakia, R.drawable.slovakia,
                       R.string.slovakia_url));
    EU.add(new Country(R.string.slovenia, R.drawable.slovenia,
                       R.string.slovenia_url));
    EU.add(new Country(R.string.spain, R.drawable.spain,
                       R.string.spain_url));
    EU.add(new Country(R.string.sweden, R.drawable.sweden,
                       R.string.sweden_url));
    EU.add(new Country(R.string.united_kingdom,
                       R.drawable.united_kingdom,
                       R.string.united_kingdom_url));
  }

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    setListAdapter(new CountryAdapter());
  }
```

```
@Override
protected void onListItemClick(ListView l, View v,
                               int position, long id) {
  startActivity(new Intent(Intent.ACTION_VIEW,
                           Uri.parse(getString(EU.get(position).url))));
}

static class Country {
  int name;
  int flag;
  int url;

  Country(int name, int flag, int url) {
    this.name=name;
    this.flag=flag;
    this.url=url;
  }
}

class CountryAdapter extends ArrayAdapter<Country> {
  CountryAdapter() {
    super(EU4You.this, R.layout.row, R.id.name, EU);
  }

  @Override
  public View getView(int position, View convertView,
                      ViewGroup parent) {
    CountryWrapper wrapper=null;

    if (convertView==null) {
      convertView=getLayoutInflater().inflate(R.layout.row, null);
      wrapper=new CountryWrapper(convertView);
      convertView.setTag(wrapper);
    }
    else {
      wrapper=(CountryWrapper)convertView.getTag();
    }
```

```
        wrapper.populateFrom(getItem(position));

        return(convertView);
      }
    }

    class CountryWrapper {
      private TextView name=null;
      private ImageView flag=null;
      private View row=null;

      CountryWrapper(View row) {
        this.row=row;
      }

      TextView getName() {
        if (name==null) {
          name=(TextView)row.findViewById(R.id.name);
        }

        return(name);
      }

      ImageView getFlag() {
        if (flag==null) {
          flag=(ImageView)row.findViewById(R.id.flag);
        }

        return(flag);
      }

      void populateFrom(Country nation) {
        getName().setText(nation.name);
        getFlag().setImageResource(nation.flag);
      }
    }
  }
```

Les Figures 36.2, 36.3 et 36.4 montrent l'aspect de cette activité sur un émulateur HVGA classique, un émulateur WVGA et un émulateur QVGA.

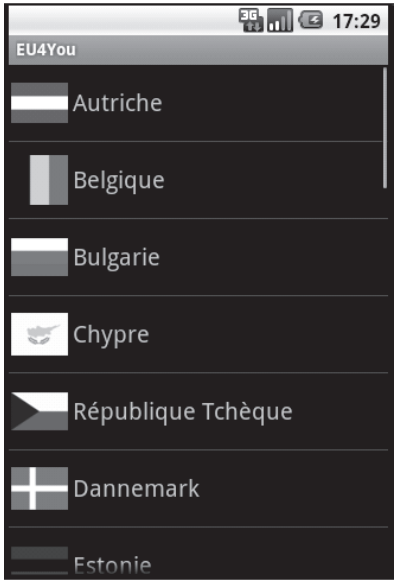


Figure 36.2
EU4You, version initiale, écran HVGA.

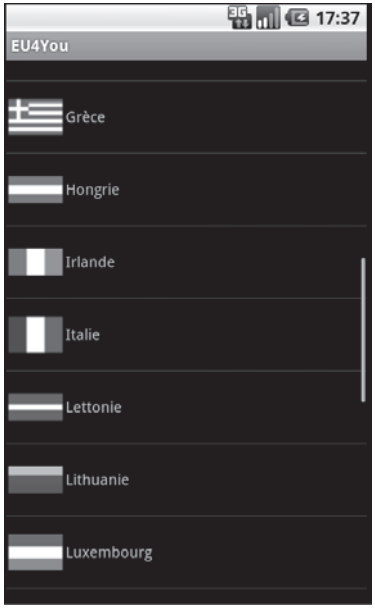


Figure 36.3
EU4You, version initiale, écran WVGA (800 × 400 pixels).

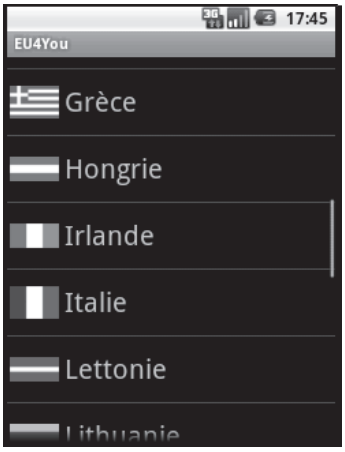


Figure 36.4
EU4You, version initiale, écran QVGA.

Corriger les polices

Le premier problème à corriger est celui de la taille de la police. Comme vous pouvez le constater, avec une taille fixe de 20 pixels les fontes vont de très grande à minuscule selon la taille et la densité de l'écran. Sur un écran WVGA, le texte peut même être assez difficile à lire.

Nous pourrions considérer la dimension de la police comme une ressource `res/values/dimens.xml` et utiliser différentes versions de celle-ci en fonction de la taille ou de la densité de l'écran, mais il est plus simple de préciser une taille indépendante de la densité – `5mm`, par exemple. C'est ce que nous avons fait dans le projet `ScreenSizes/EU4You_2` :

```
<?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:padding="2dip"
                    android:minHeight="?android:attr/listPreferredItemHeight">
  <ImageView android:id="@+id/flag"
             android:layout_width="wrap_content"
             android:layout_height="wrap_content"
             android:layout_gravity="center_vertical|left"
             android:paddingRight="4px"
  />
  <TextView android:id="@+id/name"
```

```
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_gravity="center_vertical|right"
                android:textSize="5mm"
    />
</LinearLayout>
```

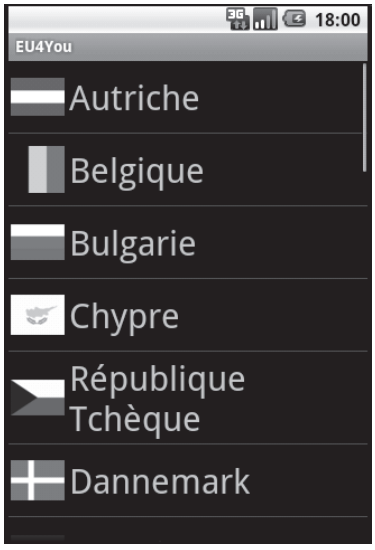


Figure 36.5
EU4You, police de 5 mm, écran HVGA.

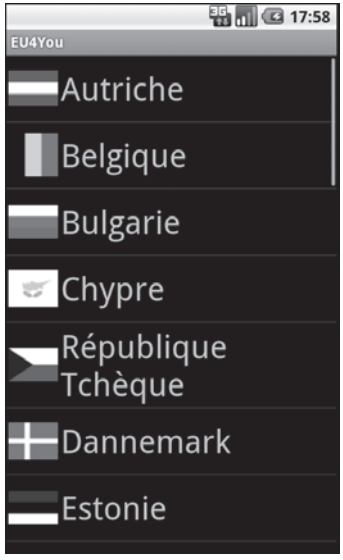


Figure 36.6
EU4You, police de 5 mm, écran WVGA (800 × 400 pixels).

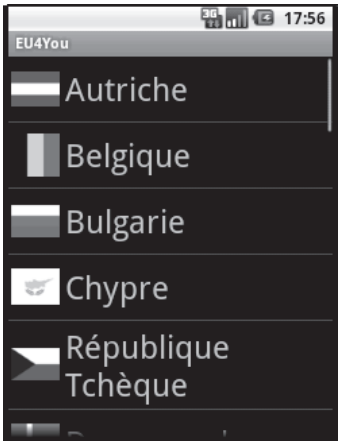


Figure 36.7
EU4You, police de 5 mm, écran QVGA.

Comme le montrent les Figures 36.5, 36.6 et 36.7, la police a désormais toujours la même taille et elle est suffisamment grande pour s'accorder avec les drapeaux.

Corriger les icônes

Comment faire pour les icônes ? Leur taille devrait varier également or ce sont les mêmes pour les trois émulateurs.

Cependant, Android adapte automatiquement la taille des ressources bitmap, même avec `<supports-screens>` et ses attributs à `true`. L'avantage est que cela signifie que vous pouvez vous contenter de laisser ces bitmaps comme ils sont. Par contre, ceci signifie que c'est le terminal qui devra adapter la taille, ce qui a évidemment un coût en terme de temps CPU (et donc en terme de batterie). En outre, les algorithmes de redimensionnement peuvent ne pas être optimaux comparé à ce que vous pouvez faire avec les outils graphiques de votre machine de développement.

Le projet `ScreenSizes/EU4You_3` utilise donc deux répertoires, `res/drawable-ldpi` et `res/drawable-hdpi`, pour y placer des versions respectivement plus petites et plus grandes des drapeaux. Il renomme également `res/drawable` en `res/drawable-mdpi`. Android utilisera ainsi les tailles de drapeaux adaptées à la densité de l'écran du terminal ou de l'émulateur.

Utilisation de l'espace

Bien que l'activité s'affiche correctement sur un écran WVGA en mode portrait, elle gaspille énormément d'espace en mode paysage, comme le montre la Figure 36.8.

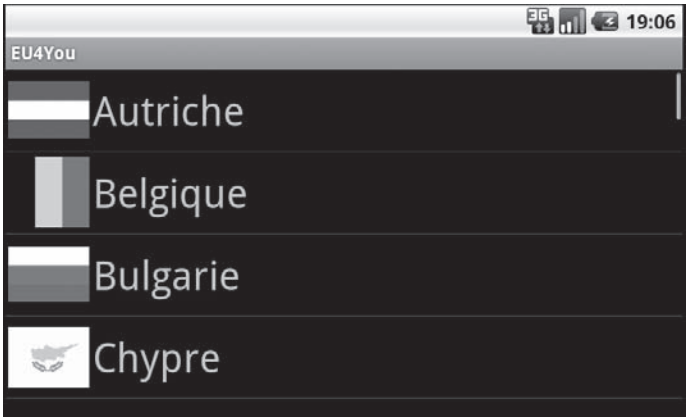


Figure 36.8
EU4You, écran WVGA (800 × 400 pixels) en mode paysage.

Au lieu de lancer une activité séparée pour exécuter le navigateur web, vous pouvez faire apparaître le contenu Wikipedia directement dans l'activité principale lorsque le terminal a un écran large et qu'il est en mode paysage.

Pour ce faire, nous devons d'abord cloner le layout `main.xml` dans un répertoire `res/layout-large-land` et lui ajouter un widget `WebView`. C'est ce que nous faisons dans `ScreenSizes/EU4You_4` :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="fill_parent"
              android:layout_height="fill_parent" >
  <ListView android:id="@android:id/list"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent"
            android:layout_weight="1"
  />
  <WebView android:id="@+id/browser"
           android:layout_width="fill_parent"
           android:layout_height="fill_parent"
           android:layout_weight="1"
  />
</LinearLayout>
```

Puis il faut modifier notre activité pour qu'elle recherche la présence de ce `WebView` et qu'elle l'utilise s'il est présent ; sinon on lance une activité de navigation classique :

```
@Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

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

    browser=(WebView)findViewById(R.id.browser);

    setListAdapter(new CountryAdapter());
  }

  @Override
  protected void onListItemClick(ListView l, View v,
                                 int position, long id) {
    String url=getString(EU.get(position).url);

    if (browser==null) {
      startActivity(new Intent(Intent.ACTION_VIEW,
                               Uri.parse(url)));
    }
    else {
      browser.loadUrl(url);
    }
  }
```

Ceci nous donne une version utilisant plus efficacement l'espace disponible, comme le montre la Figure 36.9.

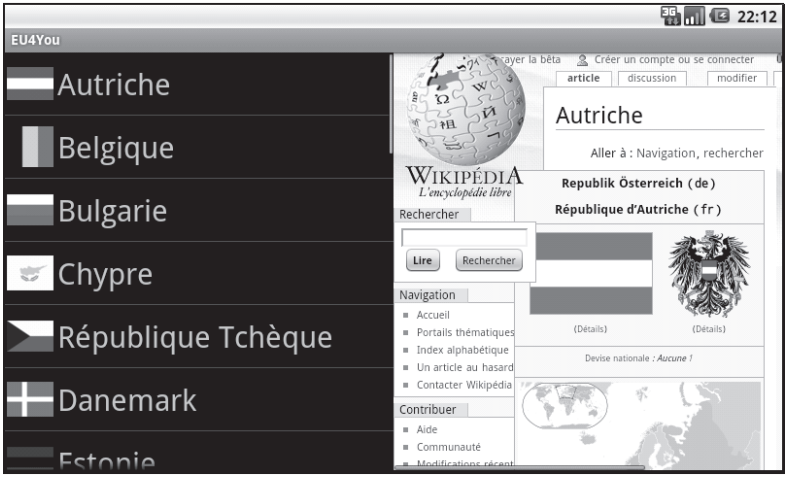


Figure 36.9
EU4You, écran WVGA (800 × 400 pixels) en mode paysage configuré en densité normale et montrant le `WebView` *intégré.*

Si l'utilisateur clique sur un lien de la page Wikipedia, le navigateur s'ouvre afin de faciliter la navigation.

Notez que, pour tester la version de cette activité et son nouveau comportement, vous devez configurer correctement l'émulateur. Par défaut, en effet, Android configure les terminaux WVGA en haute densité, ce qui signifie que WVGA n'est pas "grand" du point de vue des ensembles de ressources, mais `normal`. Vous devrez donc créer un nouvel AVD en le configurant avec une densité normale (`medium`) afin d'obtenir une taille d'écran `large`.

Et si ce n'est pas un navigateur ?

Bien sûr, EU4You triche un peu. Ici, la seconde activité est un navigateur (ou un `WebView` dans la forme intégrée) : ce n'est pas votre propre création. Les choses se compliquent un peu lorsque la seconde activité est une activité à vous, avec plusieurs widgets configurés dans un layout, et que vous souhaitez à la fois l'utiliser comme une activité (sur un petit écran) et l'intégrer dans l'activité principale de l'interface utilisateur (pour les écrans plus grands).

Il existe un motif de conception pour ce type de scénario :

1. Développez et testez d'abord la seconde activité comme une activité autonome.
2. Faites en sorte que toutes les méthodes du cycle de vie de la seconde activité délèguent leur traitement à une classe interne. Déplacez toutes les données membres de l'activité qui ne servent qu'à la classe interne dans celle-ci et assurez-vous que cela fonctionne.
3. Extrayez la classe interne pour en faire une classe publique séparée et assurez-vous que cela fonctionne.
4. Pour votre première activité (principale), créez un layout pour les grands écrans et utilisez la directive `<include>` pour y intégrer le contenu du layout de la seconde activité au bon endroit.
5. Si le layout de la seconde activité a été trouvé dans la première activité (en vérifiant, par exemple, l'existence d'un widget particulier à l'aide de `findViewById()`), créez une instance de la classe publique de l'étape 3 et faites-lui gérer tous ces widgets. Modifiez votre code pour qu'il désigne directement cette classe plutôt que lancer la seconde activité comme dans la section précédente.

En résumé, utilisez une classe publique et un layout réutilisable afin de conserver au même endroit votre code et vos ressources tout en les utilisant à la fois comme une activité autonome et comme partie d'une version grand écran d'une activité principale.

Nos amis ont-ils quelques bogues ?

Le DROID de Motorola, fourni avec Android 2.0, comprenait deux bogues concernant les tailles d'écrans :

- Ses valeurs pour la densité de l'écran étaient incorrectes, à la fois en horizontal et en vertical : ceci signifie qu'il utilisait des dimensions incorrectes pour les tailles physiques `pt`, `mm` et `in`.
- Il utilisait Android 2.0 au niveau 6 de l'API au lieu de 5 : les répertoires de ressources spécifiques aux versions devaient donc utiliser le suffixe `-v6` au lieu de `-v5`.

Ces deux bogues ont été corrigés par Android 2.0.1 et aucun autre terminal ne devrait désormais être fourni avec Android 2.0 ou être affecté par ces bogues.

37

Gestion des terminaux

Pour les constructeurs de terminaux, Android est gratuit car c'est un projet open-source. Ils ont donc carte blanche pour en faire ce qu'ils veulent sur leurs matériels. Ceci a pour effet d'offrir un très grand choix aux utilisateurs, qui peuvent ainsi disposer de terminaux Android de toutes les formes, tailles ou couleurs. Ceci signifie également que les développeurs devront tenir compte des différences et des particularités des terminaux.

Ce chapitre vous donnera quelques conseils pour gérer les problèmes spécifiques des terminaux, qu'il faut ajouter à ce que nous avons expliqué sur la taille des écrans au Chapitre 36.

Cette application contient des instructions explicites

Au départ, le G1 était le seul terminal Android disponible : lorsque l'on écrivait une application Android, on pouvait donc supposer la présence d'un clavier physique alphanumérique, d'un trackball pour la navigation, etc. Depuis, d'autres terminaux ont vu le jour (le HTC Magic, par exemple), avec des équipements matériels différents (pas de clavier physique, notamment).

Dans l'idéal, votre application peut fonctionner sans tenir compte de la présence des différents composants matériels. Certaines applications, cependant, ne pourront pas marcher si le terminal n'a pas certaines caractéristiques. Un jeu plein écran, par exemple, peut dépendre d'un clavier physique ou d'un trackball pour signaler les actions de l'utilisateur – les claviers logiciels et les écrans tactiles peuvent ne pas suffire.

Heureusement, à partir d'Android 1.5, vous pouvez ajouter des instructions explicites indiquant vos besoins à Android, afin que votre application ne puisse pas s'installer sur les terminaux qui ne disposent pas du matériel nécessaire.

Outre l'identifiant système indiquant la version à laquelle s'adresse votre projet, vous pouvez désormais ajouter des éléments `<uses-configuration>` à l'élément `<manifest>` du fichier `AndroidManifest.xml` afin de préciser le matériel requis par votre application. Chacun de ces éléments permet de décrire une configuration matérielle valide pour votre application. Actuellement, vous pouvez préciser cinq exigences matérielles :

- `android:reqFiveWayNav`. Indique si l'on a besoin d'un dispositif de pointage avec cinq directions (`android:reqFiveWayNav = "true"`, par exemple).
- `android:reqNavigation`. Restreint le type du dispositif de pointage à cinq directions (`android:reqNavigation = "trackball"`, par exemple).
- `android:reqHardKeyboard`. Indique si l'on a besoin d'un clavier physique (`android:reqHardKeyboard = "true"`, par exemple).
- `android:reqKeyboardType`. Combiné avec `android:reqHardKeyboard`, indique un type de clavier physique particulier (`android:reqKeyboardType = "qwerty"`, par exemple).
- `android:reqTouchScreen`. Indique le type d'écran tactile éventuellement nécessaire (`android:reqTouchScreen = "finger"`, par exemple).

Android 1.6 ajoute un élément similaire, `<uses-feature>`, conçu pour documenter les besoins d'une application concernant d'autres fonctionnalités facultatives des terminaux Android. Cet élément peut contenir les attributs suivants :

- `android:glEsVersion`. Indique que votre application a besoin d'OpenGL. La valeur de cet attribut précise la version minimale d'OpenGL (`0x00010002` pour OpenGL 1.2 ou supérieur, par exemple).
- `android:name = "android.hardware.camera"`. Indique que votre application a besoin d'un appareil photo.
- `android:name = "android.hardware.camera.autofocus"`. Indique que votre application a besoin d'un appareil photo autofocus.

Boutons, boutons, qui a des boutons ?

La présence des boutons sur les différents terminaux Android est très variable : boutons physiques ou boutons logiciels sur l'écran, voire pas de bouton du tout.

Le HTC Dream (l'autre nom du G1), par exemple, a des boutons appel, fin d'appel, home, retour, menu et appareil photo, ainsi qu'un contrôle du volume et un bouton réservé aux recherches sur son clavier physique. Sur le HTC Magic (également appelé myTouch 3G), le bouton pour l'appareil photo a été remplacé par le bouton de recherche. La tablette Android Archos 5 n'a aucun bouton physique hormis le contrôle du volume : les boutons home, retour et menu sont tous logiciels.

Vous devez donc faire attention lorsque vous supposez l'existence ou l'emplacement des boutons physiques. Proposez d'autres moyens d'effectuer les opérations réservées aux boutons : si, par exemple, vous redéfinissez le contrôle du volume pour qu'il serve de touche Page vers le haut/vers le bas, assurez-vous que l'utilisateur aura un autre moyen de faire défiler les pages.

Un marché garanti

Comme on l'a mentionné dans l'introduction de ce chapitre, Android est open-source. Plus précisément, il est essentiellement placé sous les termes de l'Apache Software License 2.0. Cette licence imposant peu de restrictions aux constructeurs de terminaux, il est donc possible que l'un d'eux crée un terminal qui, franchement, ne conviendra pas à Android. Les applications fournies avec le terminal s'exécuteront correctement, mais pas les applications tierces comme celles que vous pourriez écrire.

Pour aider à résoudre ce problème, Google a créé quelques applications, comme l'Android Market, qui ne sont pas open-source. Bien qu'elles soient disponibles pour les constructeurs de terminaux, ceux qui proposent l'Android Market sont d'abord testés afin d'assurer que les utilisateurs pourront utiliser correctement leurs terminaux.

Un ingénieur de Google citait le cas d'un constructeur prêt à produire un téléphone avec un écran QVGA avant la sortie d'Android 1.6, qui a officiellement ajouté le support QVGA à cette plateforme. Même si ce constructeur s'était arrangé pour que les applications fournies avec l'appareil fonctionnent sur un écran de résolution réduite, c'était l'horreur pour les applications tierces. Apparemment, Google a refusé de fournir Android Market à ce constructeur pour ce produit.

Par conséquent, l'existence même d'Android Market sur un terminal, outre qu'il offre un moyen de distribution de vos applications, sert également un peu de certificat de conformité d'un terminal en garantissant qu'il devrait supporter les applications tierces correctement écrites.

Les détails scabreux

Malheureusement, l'Android Market ne garantit pas un déploiement dépourvu de tout problème sur les terminaux sur lesquels il est installé ; il n'empêche pas non plus les constructeurs de proposer des terminaux qui ne passent pas par le Market. Inévitablement, certains auront donc des particularités pouvant avoir un impact négatif sur vos applications. Voici une liste de quelques terminaux Android, dans l'ordre de leur venue sur le marché, et leurs différences par rapport à des dispositifs plus standard.

Tablette Internet Android Archos 5

La tablette Internet Android Archos 5 est le premier dispositif grand public reposant exclusivement sur le projet open-source Android. À la différence des téléphones HTC, Motorola et autres, l'Archos 5 n'est pas un terminal certifié par Google et ne dispose donc ni de l'Android Market, ni de Google Maps, ni des autres applications propriétaires de Google.

L'Archos 5 est un terminal WVGA, mais il est livré avec Android 1.5 : de base, il ne reconnaît donc pas la nouvelle désignation `-large` pour les ensembles de ressources (voir Chapitre 36). Ce terminal n'ayant pas été beaucoup vendu, vous devrez vous contenter de son interface non optimisée jusqu'à ce que le support d'Android 1.6 lui soit ajouté.

L'écran tactile de l'Archos 5 est résistif et non capacitif : pour manipuler l'écran, les utilisateurs doivent donc utiliser leurs ongles ou un stylet plutôt que leurs doigts. Vous devez en tenir compte lorsque vous développez des interfaces tactiles.

L'Archos 5, depuis son firmware 1.1.01, renvoyait une valeur incorrecte pour `ANDROID_ID` (un identifiant unique affecté à chaque terminal Android). Dans l'émulateur, `ANDROID_ID` vaut `null` et est censé être une chaîne hexadécimale sur les vrais terminaux or, avec l'Archos 5, cet identifiant n'est ni l'un ni l'autre. Si vous testez simplement s'il vaut `null` ou non, vous n'aurez donc pas de problème ; en revanche, si vous attendez une valeur hexadécimale, ce comportement vous posera quelques problèmes.

L'Archos 5 n'étant pas un téléphone, les fonctionnalités liées à la téléphonie – les appels *via* `ACTION_DIAL`, par exemple – ne sont pas disponibles. Il n'a pas non plus d'appareil photo. Comme on l'a déjà mentionné, il ne dispose pas de Google Maps, de l'Android Market et des autres applications propriétaires de Google.

Notons également que l'IMEI de l'Archos est un faux puisque ce n'est pas un téléphone.

CLIQ/DEXT de Motorola

Le CLIQ de Motorola (appelé DEXT en dehors des États-Unis) est un terminal HVGA livré avec Android 1.5.

Le CLIQ possède un pad directionnel pour la navigation non tactile. Toutefois, ce pad se trouve sur un clavier alphanumérique coulissant et n'est donc pas disponible lorsque le téléphone est en mode portrait – à moins de forcer le mode portrait *via* le manifeste de votre activité et d'obliger les utilisateurs à utiliser leur CLIQ avec le clavier sorti. N'écrivez pas des applications qui supposent qu'un terminal dispose toujours d'un pad directionnel.

Le CLIQ est également fourni avec MOTOBLUR, la couche de présentation sociale de Motorola. Ceci signifie que l'écran d'accueil, les contacts et d'autres fonctionnalités d'Android ont été remplacés par ceux spécifiques à MOTOBLUR. Ceci ne devrait pas poser trop de problème si vous vous en tenez au SDK. La seule conséquence un peu particulière est que vous n'aurez pas accès à tous les contacts MOTOBLUR *via* le fournisseur de contenu Android Contacts – les contacts Facebook, par exemple, sont accessibles à MOTOBLUR mais pas aux applications tierces, probablement pour des raisons de licence. Cette situation peut être amenée à changer lorsque le CLIQ sera mis à jour avec le nouveau système `ContactsContract` d'Android 2.0.1.

DROID/Milestone de Motorola

Le DROID de Motorola (appelé Milestone en dehors des États-Unis) est un terminal WVGA854 livré avec Android 2.0, bien que la plupart des terminaux vendus aient depuis été mis à jour avec Android 2.0.1.

Le DROID, comme le CLIQ, possède un pad directionnel sur un clavier amovible, ce qui signifie que le pad n'est pas accessible lorsque le téléphone est en mode portrait.

Le DROID ayant un écran WVGA854 de taille normale, Android considérera qu'il a un écran haute densité et utilisera donc les ensembles de ressources `-hdpi`.

Nexus One de Google/HTC

Le Nexus One construit par HTC et vendu par Google est un terminal WVGA800 livré avec Android 2.1.

Comme le DROID, l'écran du Nexus One est considéré comme ayant une haute densité (`-hdpi`).

BACKFLIP de Motorola

Le BACKFLIP de Motorola utilise une autre approche pour le pointage : plutôt qu'un trackball ou un pad directionnel, il dispose de deux options de navigation non tactile :

- Son clavier physique a des touches directionnelles comme celui d'un PC, qui produisent des événements `DPAD` standard.
- Le touchpad qui se trouve au dos de l'écran tactile produit des événements trackball (ou `DPAD` si les précédents ne sont pas consommés).

38

Gestion des changements de plateformes

Android continuera d'évoluer rapidement au cours des prochaines années. À un moment, le rythme des changements déclinera sans doute mais, pour l'instant, vous devez supposer que des versions importantes d'Android verront le jour tous les 6 ou 12 mois et modifieront la gamme des matériels. Par conséquent, bien qu'Android cible actuellement les téléphones, vous verrez bientôt apparaître des tablettes Android, des lecteurs multimédias Android, etc.

La plupart de ces changements auront peu d'impact sur le code existant. Toutefois, certains nécessiteront au moins quelques tests de vos applications et, éventuellement, des modifications de votre code en fonction des résultats de ces tests.

Ce chapitre présente les domaines qui peuvent poser problème à mesure qu'Android évolue, ainsi que quelques conseils pour les résoudre.

Gestion de la marque

À l'heure où ce livre est écrit, les terminaux Android existants sont des téléphones Google. Ceci signifie qu'ils disposent de l'interface standard d'Android – que vous retrouvez dans l'émulateur – et qu'ils intègrent tout un lot d'applications comme Google Maps et Gmail. De leur côté, les constructeurs sont autorisés à placer le logo "with Google" sur leurs produits, mais ce n'est pas toujours le cas.

Certains constructeurs se servent également d'Android comme d'une base et modifient ce qu'il contient, éventuellement en ajoutant leurs propres applications et en changeant l'aspect de l'interface (icônes des menus, structure de l'écran d'accueil, etc).

D'autres peuvent utiliser uniquement le dépôt open-source d'Android et, bien qu'ils proposent l'aspect standard, ne pas disposer des applications propriétaires.

Même aujourd'hui, certains terminaux proposent un mélange d'applications en fonction des régions où ils sont distribués : les possesseurs américains du G1, par exemple, disposent d'une application Amazon MP3 Store qui n'est pas disponible dans tous les pays.

Une application indépendante de tous ces critères devrait pouvoir s'exécuter n'importe où. En revanche, si votre code ou votre documentation suppose l'existence de Google Maps, Gmail, Amazon MP3 Store, etc., des problèmes surgiront inévitablement. Assurez-vous de tester soigneusement votre code dans les environnements où ces applications ne sont pas disponibles.

Des choses qui risquent de vous rendre nerveux

La plupart des éléments cités dans la section précédente concernaient les changements matériels. Examinons maintenant les problèmes que peuvent vous poser les modifications du système d'exploitation lui-même.

Hiérarchie des vues

Android n'est pas conçu pour gérer des hiérarchies de vues complexes. Ici, "hiérarchie de vues" signifie conteneurs contenant des conteneurs contenant des conteneurs contenant des widgets.

Comme le montre la Figure 38.1, le programme *Hierarchy Viewer* décrit au Chapitre 35 représente bien ces hiérarchies. Dans cet exemple, on voit une hiérarchie sur cinq niveaux car la longueur de la plus longue chaîne de conteneurs et de widgets (de `PhoneWindow$DecorView` à `Button`) a cinq nœuds.

Android a toujours limité la profondeur maximale des hiérarchies de vues. Toutefois, Android 1.5 a réduit cette limite : certaines applications qui fonctionnaient parfaitement

avec Android 1.1 échouaient donc en produisant une exception `StackOverflowException` dans la nouvelle version. C'était, bien sûr, assez frustrant pour les développeurs qui ne se rendaient pas compte que le problème provenait de la profondeur de la hiérarchie et qui se faisaient piéger par cette modification.

Les leçons à tirer de cette expérience sont les suivantes :

- Limitez la profondeur de vos hiérarchies de vues. Si vous atteignez une profondeur à deux chiffres, vous risquez fortement d'épuiser l'espace alloué à la pile.
- Si vous rencontrez une exception `StackOverflowException` et que la trace montre qu'elle survient quelque part au milieu du dessin de vos widgets, c'est que votre hiérarchie de vues est probablement trop complexe.

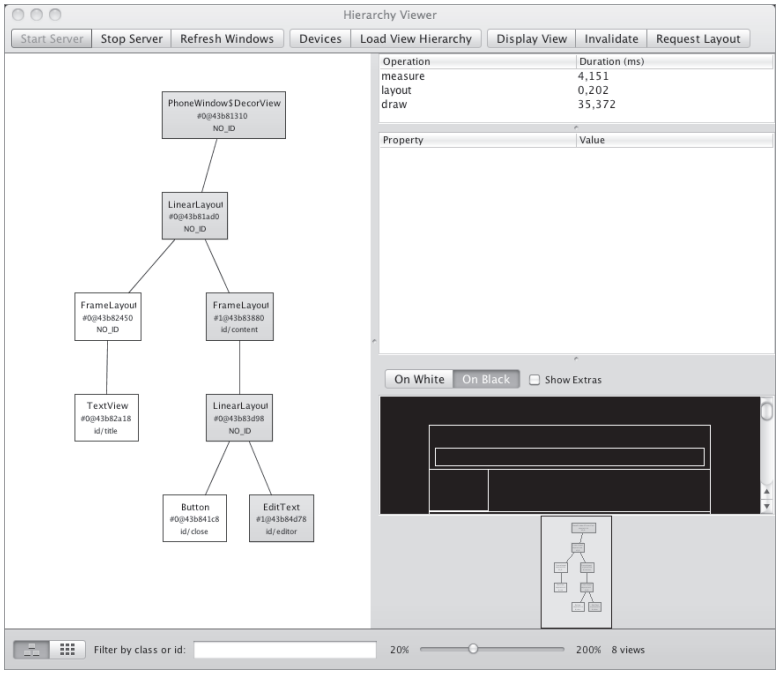


Figure 38.1
Vue d'un layout dans le Hierarchy Viewer.

Changement des ressources

L'équipe d'Android peut modifier les ressources au cours d'une mise à jour, ce qui peut avoir des effets inattendus pour votre application. Avec Android 1.5, par exemple, le fond du widget `Button` a été modifié afin de permettre de créer des boutons plus petits. Les applications qui se fiaient à la taille minimale précédente ont donc dû revoir leurs interfaces utilisateurs.

De même, les applications peuvent réutiliser des ressources publiques – des icônes, par exemple – qu'elles puisent directement dans Android. Bien que procéder ainsi permette d'économiser l'espace de stockage, nombreuses sont ces ressources qui sont publiques par nécessité et ne sont pas considérées comme faisant partie du SDK. Les constructeurs, par exemple, peuvent modifier les icônes pour qu'elles correspondent à l'aspect de leur interface propriétaire : penser que ces icônes auront toujours le même aspect est donc un peu dangereux. Il est préférable de prendre ces ressources dans le dépôt du projet Android open-source (**http://source.android.com/**) et de les copier dans votre propre base de code.

Gestion des modifications de l'API

L'équipe d'Android a généralement fait un bon travail en maintenant la stabilité des API et en utilisant un modèle d'obsolescence lorsqu'elle les modifie. Avec Android, être obsolète n'implique pas d'être supprimé – l'utilisation est simplement déconseillée. De nouvelles API sont fournies à chaque mise à jour d'Android et les modifications des API existantes sont bien documentées dans les rapports qui accompagnent la nouvelle version.

Malheureusement, l'Android Market (le canal de distribution principal des applications Android) ne permet de ne déposer qu'un seul fichier APK par application. Ceci signifie donc que le même fichier APK doit pouvoir s'accommoder du plus grand nombre possible de versions. Souvent, votre code "fonctionne" et ne nécessite aucune modification ; parfois, cependant, vous devrez faire quelques changements, surtout si vous voulez prendre en compte les nouvelles API sans remettre en cause les anciennes versions. Nous allons présenter quelques techniques permettant de traiter ces situations.

Détecter la version d'Android

Si vous voulez créer plusieurs branches dans votre code en fonction de la version d'Android sur laquelle il s'exécute, le plus simple consiste à utiliser `android.os.VERSION.SDK_INT`. Cette valeur entière publique et statique contient le même niveau d'API que celui que vous utilisez lorsque vous créez des AVD et que vous précisez dans le manifeste. Vous pouvez donc la comparer à `android.os.VERSION_CODES.DONUT`, par exemple, pour savoir si l'application s'exécute sur Android 1.6 ou une version plus récente.

Envelopper l'API

Tant que les API que vous tentez d'utiliser existent sur toutes les versions d'Android que vous supportez, vous pouvez vous contenter d'effectuer des branchements dans votre code. Les choses se compliquent lorsque les API changent – lorsque les méthodes prennent de nouveaux paramètres, lorsqu'il y a de nouvelles méthodes, voire de nouvelles classes.

Vous devez écrire du code qui fonctionnera quelle que soit la version d'Android tout en tirant parti des nouvelles API lorsqu'elles sont disponibles.

Pour résoudre ce problème, une astuce consiste à utiliser l'introspection plus un soupçon de mise en cache.

Au Chapitre 8, par exemple, nous avions utilisé `getTag()` et `setTag()` pour associer un objet quelconque à une `View`. Plus précisément, nous les utilisions pour associer un objet enveloppe qui recherchait tous les widgets nécessaires lorsqu'il y en avait besoin. Nous avons également vu que les nouvelles versions de `getTag()` et `setTag()` sont indexées et prennent un identifiant de ressource en paramètre.

Cependant, ces nouvelles méthodes indexées n'existaient pas avec Android 1.5. Si vous voulez utiliser cette nouvelle technique, vous devrez imposer une version d'Android 1.6 ou supérieure ou utiliser l'introspection. Avec Android 1.5, en effet, vous pourriez associer un `ArrayList<Object>` comme marqueur et faire en sorte que votre propre paire `getTag()/setTag()` prenne l'indice de chacun de vos éléments présents dans l'ArrayList.

À titre d'exemple, étudions le projet `APIVersions/Tagger`. Notre activité a un layout simple, formé uniquement d'un widget `TextView` :

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

Le code source de l'activité `Tagger` examine la version de l'API utilisée et délègue les opérations `getTag()` et `setTag()` aux versions natives indexées (pour Android 1.6 et les versions supérieures) ou aux versions originales non indexées `getTag()` et `setTag()`, auquel cas nous utilisons un `HashMap` pour mémoriser les différents objets indexés :

```
package com.commonsware.android.api.tag;

import android.app.Activity;
import android.os.Build;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.TextView;
```

```
import java.util.HashMap;
import java.util.Date;

public class Tagger extends Activity {
  private static final String LOG_KEY="Tagger";

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

    TextView view=(TextView)findViewById(R.id.test);

    setTag(view, R.id.test, new Date());

    view.setText(getTag(view, R.id.test).toString());
  }

  public void setTag(View v, int key, Object value) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.DONUT) {
      v.setTag(key, value);
    } else {
      HashMap<Integer, Object> meta = (HashMap<Integer, Object>)v.getTag();

      if (meta==null) {
        meta=new HashMap<Integer, Object>();
      }

      meta.put(key, value);
    }
  }

  public Object getTag(View v, int key) {
    Object result=null;

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.DONUT) {
      result=v.getTag(key);
    } else {
      HashMap<Integer, Object> meta=(HashMap<Integer, Object>)v.getTag();

      if (meta==null) {
        meta=new HashMap<Integer, Object>();
      }
```

```
        result=meta.get(key);
      }
      return(result);
    }
  }
```

Si nous compilons ce code et que nous le déployions sur Android 1.6 ou une version supérieure, tout fonctionnera correctement : l'activité affichera l'heure courante.

Si nous le compilons et que nous le déployions sur Android 1.5, par contre, son exécution provoquera une exception `VerifyError`. Ici, `VerifyError` signifie en fait que nous faisons appel à des éléments qui n'existent pas dans notre version d'Android :

- Nous faisons référence à `SDK_INT`, qui n'existe pas avant Android 1.6.
- Nous faisons référence aux versions indexées de `getTag()` et `setTag()` : même si nous n'exécuterons pas la partie du code en question, le chargeur de classes cherche quand même à résoudre ces méthodes et échoue.

Nous devons donc utiliser l'introspection.

Examinons le projet `APIVersions/Tagger2`, qui utilise le même layout que le précédent, mais dont le code Java est plus élaboré :

```
package com.commonsware.android.api.tag;

import android.app.Activity;
import android.os.Build;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.TextView;
import java.lang.reflect.Method;
import java.util.HashMap;
import java.util.Date;

public class Tagger extends Activity {
  private static final String LOG_KEY="Tagger";
  private static Method _setTag=null;
  private static Method _getTag=null;

  static {
    int sdk = new Integer(Build.VERSION.SDK).intValue();

    if (sdk >= 4) {
      try {
```

```
      _setTag=View.class.getMethod("setTag",
                      new Class[] {Integer.TYPE, Object.class });
      _getTag=View.class.getMethod("getTag",
                      new Class[] {Integer.TYPE});
    } catch (Throwable t) {
      Log.e(LOG_KEY, "Could not initialize 1.6 accessors", t);
    }
  }
};

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

  TextView view=(TextView)findViewById(R.id.test);

  setTag(view, R.id.test, new Date());

  view.setText(getTag(view, R.id.test).toString());
}

public void setTag(View v, int key, Object value) {
  if (_setTag!=null) {
    try {
      _setTag.invoke(v, key, value);
    } catch (Throwable t) {
      Log.e(LOG_KEY, "Could not use 1.6 setTag()", t);
    }
  } else {
    HashMap<Integer, Object> meta=(HashMap<Integer, Object>)v.getTag();

    if (meta==null) {
      meta=new HashMap<Integer, Object>();
      v.setTag(meta);
    }

    meta.put(key, value);
  }
}

public Object getTag(View v, int key) {
  Object result=null;
```

```
        if (_getTag!=null) {
          try {
            result=_getTag.invoke(v, key);
          } catch (Throwable t) {
            Log.e(LOG_KEY, "Could not use 1.6 getTag()", t);
          }
        } else {
          HashMap<Integer, Object> meta=(HashMap<Integer, Object>)v.getTag();

          if (meta==null) {
            meta=new HashMap<Integer, Object>();
            v.setTag(meta);
          }

          result=meta.get(key);
        }
        return(result);
      }
    }
```

Quand la classe est chargée, les méthodes d'initialisation statiques s'exécutent d'abord. Ici, nous vérifions la version d'Android utilisée à l'aide de l'ancienne chaîne `SDK` et non plus avec l'entier `SDK_INT` plus récent. Si nous sommes sous Android 1.6 ou une version supérieure, nous utilisons l'introspection pour tenter de trouver les méthodes `getTag()` et `setTag()` indexées et nous mettons les résultats en cache : ces méthodes ne changeant pas au cours de l'exécution de l'application, nous pouvons les mémoriser sans crainte dans des variables statiques.

Puis, quand il faut vraiment utiliser `getTag()` ou `setTag()`, nous vérifions si les objets `Method` en cache existent ou valent `null`. Dans ce dernier cas, nous supposons qu'il faut utiliser les anciennes versions de ces méthodes. Si ces objets `Method` existent, nous nous en servons pour tirer parti des versions indexées natives.

Cette version de l'application fonctionne très bien avec Android 1.5 et les versions supérieures. À partir d'Android 1.6, le code utilisera les méthodes indexées prédéfinies alors qu'Android 1.5 utilisera notre fausse version de ces méthodes.

L'utilisation de l'introspection à l'aide de `Method` a un léger coût en termes de performances, mais elle peut être justifiée dans certains cas afin d'accéder aux API des nouvelles versions d'Android au lieu de nous restreindre aux anciennes. Nous pouvons même utiliser cette technique lorsque des classes ont été ajoutées aux nouvelles versions d'Android (voir **http://android-developers.blogspot.com/2009/04/backward-compatibility-foran-droid.html**).

39

Pour aller plus loin

Ce livre ne couvre évidemment pas tous les sujets possibles et, bien que la ressource numéro un (en dehors de cet ouvrage) soit la documentation du SDK Android, vous aurez sûrement besoin d'informations qui se trouvent ailleurs.

Une recherche web sur le mot "android" et un nom de classe est un bon moyen de trouver des didacticiels pour une classe donnée. Cependant, n'oubliez pas que les documents écrits avant fin 2008 concernent probablement le SDK M5 et nécessiteront donc des modifications très importantes pour fonctionner correctement avec les SDK actuels.

Ce chapitre vous donnera donc quelques pistes à explorer.

Questions avec, parfois, des réponses

Il existe de nombreuses ressources anglophones et francophones.

Les groupes Google consacrés à Android (**http://groups.google.com/group**) sont les endroits officiels pour obtenir de l'aide. Trois groupes sont consacrés au SDK :

- `android-beginners` est le meilleur endroit pour poster des questions de débutant.

- `android-developers` est consacré aux questions plus compliquées ou à celles qui relèvent de parties plus exotiques du SDK.
- `android-discuss` est réservé aux discussions à bâtons rompus sur tout ce qui est lié à Android, pas nécessairement aux problèmes de programmation.

Vous pouvez également consulter :

- les didacticiels et les forums consacrés à Android sur le site **http://anddev.org** ;
- `le wiki Open Mob for Android` (**http://wiki.andmob.org**) ;
- le canal IRC `#android` sur freenode (**http://freenode.net**) ;
- les tags `android` et `android-sdk` de StackOverflow (**http//stackoverflow.com/**) ;
- le forum Android de JavaRanch (**http://www.javaranch.com/**).

Aller à la source

Le code source d'Android est désormais disponible, essentiellement pour ceux qui souhaitent améliorer le système ou jouer avec ses détails internes. Toutefois, vous pouvez aussi y trouver les réponses que vous recherchez, notamment si vous voulez savoir comment fonctionne un composant particulier.

Le code source et les ressources qui y sont liées se trouvent sur le site web du projet, **http://source.android.com**. À partir de ce site, vous pouvez :

- télécharger ou parcourir le code source ;
- signaler des bogues du système lui-même ;
- proposer des patchs et apprendre comment ces patchs sont évalués et approuvés ;
- rejoindre un groupe Google particulier pour participer au développement de la plateforme Android.

Au lieu de télécharger les gigaoctets des sources d'Android, vous pouvez utiliser Google Code Search (**http://www.google.com/codesearch**) : il suffit d'ajouter `android:package` à votre requête afin de ne rechercher que ce qui concerne Android et les projets associés.

Citons également quelques ressources francophones :

- **http://www.frandroid.com** : le premier site communautaire pour les utilisateurs et les développeurs de la plateforme Android (forum, tutoriaux, wiki, actualités).
- **http://www.poingphone.com** : forums et actualités.
- **http://www.android-france.fr** : actualités de la plateforme.
- La communauté francophone est de plus en plus dynamique et a créé le channel IRC #androidfr disponible sur freenode.

Lire les journaux

Ed Burnette, qui a écrit son propre livre sur Android, est également le gestionnaire de Planet Android (**http://www.planetandroid.com/**), un agrégateur de flux pour un certain nombre de blogs consacrés à Android. En vous abonnant à ce flux, vous pourrez ainsi surveiller un grand nombre d'articles, pas nécessairement consacrés à la programmation.

Pour surveiller plus précisément les articles liés à la programmation d'Android, faites une recherche sur le mot-clé "android" sur Dzone. Vous pouvez également vous abonner à un flux qui résume cette recherche (**http://www.dzone.com/links/feed/search/android/rss.xml**).

Index

A

aapt, outil 28
AbsoluteLayout, conteneur 133, 379
Accéléromètres 211
Activity
  classe 22, 182, 219
  élément 219
    du manifeste 13
activityCreator, script 229
ActivityManager, classe 177
Adaptateur 65
adb, programme 271, 371, 376,
addMenu(), méthode 147
add(), méthode 146, 354
addProximityAlert(), méthode 347
addSubMenu(), méthode 147
addTab(), méthode 120
AlertDialog, classe 172
AnalogClock, widget 114
android
  alphabeticShortcut, attribut 156
  anyDensity, attribut 381, 382, 388
  apiKey, attribut 351, 358
  authorities, attribut 315
autoText, attribut de widget 36
background, attribut 43
capitalize, attribut de widget 36
clickable, attribut 351
collapseColumns, propriété 60
columnWidth, propriété 74
configChanges, attribut 219
content, attribut 131
digits, attribut de widget 37
drawSelectorOnTop, propriété 71, 81
ellipsize, attribut 161
enabled, attribut 155
glEsVersion, élément 404
handle, attribut 131
horizontalSpacing, propriété 74
icon, attribut 155
id
  attribut 118, 155, 351
    de main.xml 29
imeOptions, attribut 140
inputType, attribut 136
largeScreens, attribut 381, 382, 388
layout_above, propriété 54
layout_alignBaseline, propriété 55
layout_alignBottom, propriété 54

layout_alignLeft, propriété 54
layout_alignParentBottom, propriété 53
layout_alignParentLeft, propriété 53
layout_alignParentRight, propriété 53
layout_alignParentTop, propriété 53
layout_alignRight, propriété 54
layout_alignTop, propriété 54
layout_below, propriété 54
layout_centerHorizontal, propriété 53
layout_centerInParent, propriété 53
layout_centerVertical, propriété 53
layout_column, propriété 59
layout_gravity, propriété 48
layout_height, élément 379, 389, 395, 396, 398
layout_span, propriété 59
layout_toLeftOf, propriété 54
layout_toRightOf, propriété 54
layout_weight, propriété 47
layout_width, attribut de main.xml 29
layout_width, élément 379, 389, 395, 396, 398
layout_width, propriété 47
menuCategory, attribut 155
name, attribut 315, 318, 326, 404
nextFocusDown, attribut 42
nextFocusLeft, attribut 42
nextFocusRight, attribut 42
nextFocusUp, attribut 42
normalScreens, attribut 381, 382, 388
numColumns, propriété 73
numericShortcut, attribut 156
orderInCategory, attribut 155
orientation, propriété 46
paddingLeft, propriété 48
padding, propriété 48
paquetage 21
permission, attribut 320, 327
readPermission, attribut 320
reqFiveWayNav, attribut 404
reqHardKeyboard, élément 404
reqKeyboardType, élément 404
reqNavigation, élément 404
reqTouchScreen, élément 404
screenOrientation, attribut 221
script 20
shrinkColumns, propriété 60
singleLine, attribut de widget 37
smallScreens, attribut 381, 382, 388
spacing, propriété 81
spinnerSelector, propriété 81
src, attribut de widget 35
stretchColumns, propriété 60
stretchMode, propriété 74
text
  attribut de main.xml 29
  attribut de widget 34
textColor, attribut 43
typeface, attribut 159
versionCode, attribut du manifeste 15
verticalSpacing, propriété 74
visibility, attribut 42
visible, attribut 156
windowSoftInputMode, attribut 143
writePermission, attribut 320
AndroidManifest 248
AndroidManifest.xml, fichier 9, 11, 143, 164, 198, 219, 221, 314, 318, 326, 381, 404
android.os.VERSION.SDK_INT, version de l'API 412
Android Scripting Environment (ASE) 289
animateClose(), méthode 132
animateOpen(), méthode 132
animateToggle(), méthode 132
Animations 129
apk, fichier 11
appendWhere(), méthode 269
application
  élément 314, 326
    du manifeste 12
Arborescence de répertoires 9
ArrayAdapter
  adaptateur 66, 68, 85, 97
  classe 186, 276
ArrayList, classe 276
AssetManager, classe 160

AsyncTask, classe 182, 328
AutoCompleteTextView, widget 38, 77
Auto-complétion 77
AVD (Android Virtual Device) 20

B

BaseColumns, interface 314
BeanShell, programme 285
beforeTextChanged(), méthode 79
bindService(), méthode 332
bindView(), méthode 106
BLOB (Binary Large Object) 303
Bouton accessoire 140
BroadcastReceiver
  classe 335
  interface 199, 200
Builder, classe 172
buildQuery(), méthode 269
build.xml, fichier 9
bulkInsert(), méthode 302
Bundle
  classe 194, 204, 212, 218
  objet 193
Button, widget 35

C

Calendar, classe 112
cancelAll(), méthode 338
cancel(), méthode 338
canGoBack(), méthode 167
canGoBackOrForward(), méthode 167
canGoForward(), méthode 167
Catégories d'activités 196
CheckBoxPreference, élément 247
CheckBox, widget 38
checkCallingPermission(), méthode 321
check(), méthode 41
Chronometer, widget 115
clearCache(), méthode 167
clearCheck(), méthode 41
clearHistory(), méthode 167
clear(), méthode 246
close(), méthode 132, 265, 270, 277
color, élément 238
ColorStateList, widget 43
commit(), méthode 246
CompoundButton, widget 40
Configuration, classe 383
ContentManager, classe 338
ContentObserver, classe 315
ContentProvider
  classe 307
  interface 302
ContentResolver, classe 302, 315
ContentValues, classe 266, 302, 310, 311
Context, classe 246, 277
ContextMenuInfo, classe 148
convertView, paramètre de getView() 89
createDatabase(), méthode 271
createFromAsset(), méthode 160
createFromFile(), méthode 160
createItem(), méthode 355
create(), méthode 173
createTabContent(), méthode 121
Criteria, classe 345
Cursor
  classe 267
  interface 299, 308
  widget 67
CursorAdapter, adaptateur 67, 106
CursorFactory, classe 271

D

DatabaseHelper, classe 308
DateFormat, classe 112
DatePickerDialog, widget 110
DatePicker, widget 110
DDMS (Dalvik Debug Monitor Service) 348
ddms, programme 370
DefaultHttpClient, classe 292
default.properties, fichier 9
delete(), méthode 266, 302, 312
DialogWrapper, classe 302
DigitalClock, widget 114
dimen, élément 237
dip, unité de mesure 380

doInBackground(), méthode 183
Drawable
  classe 355
  interface 231
draw(), méthode 355

E

edit(), méthode 246
EditTextPreference, élément 255
EditText, widget 36
Environment, classe 281
Espace de noms 12
execSQL(), méthode 265
execute(), méthode 183, 188, 292
ExpandableListView, classe 133

F

fill_parent, valeur de remplissage 47
findViewById (), méthode 30
findViewById(), méthode 43, 89, 120, 273, 352
findViewById(); méthode 91
finish(), méthode 193, 280
Forecast, classe 295, 329
format(), méthode 228
FrameLayout, conteneur 117, 125
fromHtml(), méthode 228

G

Gallery, widget 81
GeoPoint, classe 353
getAltitude(), méthode 345
getAsInteger(), méthode 266
getAssets(), méthode 160
getAsString(), méthode 266
getAttributeCount(), méthode 236
getAttributeName(), méthode 236
getBearing(), méthode 345
getBestProvider(), méthode 345
getBoolean(), méthode 246
getCallState(), méthode 362
getCheckedRadioButtonId(), méthode 41
getColor(), méthode 238
getColumnIndex(), méthode 270
getColumnNames(), méthode 270
getConfiguration(), méthode 383
getContentProvider(), méthode 302
getContentResolver(), méthode 315
getCount(), méthode 270
getDefaultSharedPreferences(), méthode 246, 247
getDimen(), méthode 237
getExternalStorageDirectory(), méthode 281
getInputStream(), méthode 303
getInt(), méthode 270
getItemId(), méthode 147
getLastKnownPosition(), méthode 345
getLastNonConfigurationInstance(), méthode 217
getLatitude(), méthode 293
getListView(), méthode 69
getLongitude(), méthode 293
getMapController(), méthode 352
getMenuInfo(), méthode 148
getNetworkType(), méthode 362
getOutputStream(), méthode 303
getOverlays(), méthode 354
getParent(), méthode 43
getPhoneType(), méthode 362
getPreferences(), méthode 246
getProgress(), méthode 116
getProviders(), méthode 345
getReadableDatabase(), méthode 265
getRequiredColumns(), méthode 311
getResources(), méthode 234, 273, 276
getSettings(), méthode 165, 169
getSharedPreferences(), méthode 246
getSpeed(), méthode 345
getStringArray(), méthode 239
getString(), méthode 227, 270
getSubscriberId(), méthode 362
getSystemService(), méthode 338, 344
getTableName(), méthode 269
getTag(), méthode 91, 97
getType(), méthode 313

getView(), méthode 67, 86, 89, 106, 301
getWriteableDatabase(), méthode 265
getXml(), méthode 234
goBack(), méthode 167
goBackOrForward(), méthode 167
goForward(), méthode 167
GPS (Global Positioning System) 343
GradientDrawable, classe 380
group, élément 155

H

handleMessage(), méthode 178
Handler, classe 177
hasAltitude(), méthode 345
hasBearing(), méthode 345
hasSpeed(), méthode 345
hierarchyviewer, programme 366
HorizontalScrollView, conteneur 64
htmlEncode(), méthode 230
HttpClient
  classe 324
  interface 292
HttpComponents, bibliothèque 292
HttpGet, classe 292
HttpPost, classe 292
HttpRequest, interface 292
HttpResponse, classe 292
HVGA, résolution d'écran 377

I

IBinder
  classe 325
  interface 333
ImageView
  classe 303
  widget 35
IME (Input Method Editor) 136
IMF (Input Method Framework) 135
incrementProgressBy(), méthode 116
indiquant 248
InputStream, classe 273, 277, 294
InputStreamReader, classe 277
insert(), méthode 266, 302, 310
instrumentation, élément du manifeste 12
Intent, classe 196
intent-filter, élément 198
Internationalisation (I18N) 226, 239
Interpreter, classe de BeanShell 285
isAfterLast(), méthode 270
isChecked(), méthode 38
isCollectionUri(), méthode 311, 312
isEnabled(), méthode 43
isRouteDisplayed(), méthode 352
item, élément 155, 238
ItemizedOverlay, classe 354

J

JavaScript, et WebView 165
JRuby, langage de script 289
Jython, langage de script 289

K

keytool, utilitaire 359

L

LayoutInflater, classe 87, 104
LinearLayout, conteneur 46, 84, 97, 104
ListActivity, classe 67, 184
ListAdapter, adaptateur 101
ListPreference, élément 255
ListView, widget 67, 83
loadData(), méthode 166
loadUrl(), méthode 164
Localisation (L10N) 226, 239
Location, classe 293, 345
LocationListener, classe 346
LocationManager, classe 324, 344
LocationProvider, classe 344
lock(), méthode 132

M

makeText(), méthode 172
managedQuery(), méthode 299

manifest
  élément 318
  élément racine du manifeste 11
MapActivity, classe 349, 351
MapView, classe 349, 351
Menu
  classe 146, 156
  élément 155
MenuInflater, classe 156
MenuItem, classe 146
Message, classe 178
MIME, types 307
mksdcard, programme 376
moveToFirst(), méthode 270
moveToNext(), méthode 270
MyLocationOverlay, classe 357

N

name
  attribut 226, 237, 238
  élément du manifeste 13
newCursor(), méthode 271
newTabSpec(), méthode 119
newView(), méthode 106
next(), méthode 234
nine-patch, image 380
Notification, classe 189, 338
NotificationManager, classe 338
notifyChange(), méthode 315
notify(), méthode 338

O

obtainMessage(), méthode 178
onActivityResult(), méthode 203
onBind(), méthode 325, 332
OnCheckedChangeListener, interface 51
OnClickListener
  classe 112, 205
  interface 22
OnClickListener(), méthode 175
onClick(), méthode 23
onConfigurationChanged(), méthode 219
onContextItemSelected(), méthode 148, 151
onCreateContextMenu(), méthode 148, 151
onCreate(), méthode 22, 151, 192, 194, 212, 250, 262, 276, 300, 307, 324
onCreateOptionsMenu()
  menu 151
  méthode 146, 147
onCreatePanelMenu(), méthode 147
OnDateChangedListener, classe 110
OnDateSetListener, classe 110
onDestroy(), méthode 192, 324
OnItemSelectedListener, interface 72
onListItemClick(), méthode 68, 97
onLocationChanged(), méthode 346
onOptionsItemSelected(), méthode 146, 147, 151
onPageStarted(), méthode 167
onPause(), méthode 193, 200, 280, 324, 334, 357
onPostExecute(), méthode 184
onPreExecute(), méthode 184
onPrepareOptionsMenu(), méthode 146
onProgressUpdate(), méthode 184
onRatingChanged(), méthode 97
onReceivedHttpAuthRequest(), méthode 167
onReceive(), méthode 199
onRestart(), méthode 193
onRestoreInstanceState(), méthode 194, 212
onResume(), méthode 193, 200, 250, 280, 293, 324, 334, 357
onRetainNonConfigurationInstance(), méthode 216
onSaveInstanceState(), méthode 193, 194, 200, 212, 213
onStart(), méthode 180, 193, 324
onStop(), méthode 193
onTap(), méthode 356
onTextChanged(), méthode 79
OnTimeChangedListener, classe 110
OnTimeSetListener, classe 110
onTooManyRedirects(), méthode 167
onUpgrade(), méthode 262
openFileInput(), méthode 277, 280
openFileOutput(), méthode 277, 280

open(), méthode 132
openRawResource(), méthode 273, 276
OpenStreetMap, cartographie 350
OutputStream, classe 277
OutputStreamWriter, classe 277
Overlay, classe 354
OverlayItem, classe 354

P

package, attribut du manifeste 12
parse(), méthode 298, 299
PendingIntent, classe 339, 347
permission
  élément 319
    du manifeste 12
populateDefaultValues(), méthode 311
populate(), méthode 355
postDelayed(), méthode 181
post(), méthode 181
PreferenceCategory, élément 251
PreferenceScreen, élément 247, 251
PreferencesManager, classe 246
ProgressBar, widget 115, 178
provider
  élément 314
    du manifeste 13
publishProgress(), méthode 184
px, unité de mesure 380

Q

query(), méthode 267, 308
queryWithFactory(), méthode 271
QVGA, résolution d'écran 377

R

RadioButton, widget 40
RadioGroup, widget 41
RatingBar, widget 94
rawQuery(), méthode 267
rawQueryWithFactory(), méthode 271
receiver, élément 199
registerContentObserver(), méthode 316
registerForContextMenu(), méthode 147
registerReceiver(), méthode 200
RelativeLayout, conteneur 379
reload(), méthode 167
remove(), méthode 246
removeProximityAlert(), méthode 347
removeUpdates(), méthode 347
requery(), méthode 270, 302
requestFocus(), méthode 43
requestLocationUpdates(), méthode 346
Resources
  classe 234, 273
  élément 237
ResponseHandler, classe 292
ressources, élément 226
RingtonePreference, élément 247
R.java, fichier 10
RowModel, classe 97
Runnable, classe 181
runOnUiThread(), méthode 182

S

ScrollView, conteneur 62
SecurityException, exception 318
SeekBar, widget 116
sendBroadcast(), méthode 321, 329
sendMessageAtFrontOfQueue(), méthode 178
sendMessageAtTime(), méthode 178
sendMessageDelayed(), méthode 178
sendMessage(), méthode 178
SensorManager, classe 260
Service
  classe 324
  élément 326
    du manifeste 14
ServiceConnection, classe 333
setAccuracy(), méthode 345
setAdapter(), méthode 67, 71
setAlphabeticShortcut(), méthode 147
setAltitudeRequired(), méthode 345
setBuiltInZoomControls(), méthode 353
setCenter(), méthode 353
setCheckable(), méthode 147

setChecked(), méthode 38, 42
setChoiceMode(), méthode 69
setColumnCollapsed(), méthode 60
setColumnStretchable(), méthode 60
setContent(), méthode 119, 121
setContentView(), méthode 30
setCostAllowed(), méthode 345
setCurrentTab(), méthode 120
setDefaultFontSize(), méthode 169
setDropDownViewResource(), méthode 71, 72
setDuration(), méthode 172
setEnabled(), méthode 156
setFantasyFontFamily(), méthode 169
setFlipInterval(), méthode 129
setGravity(), méthode 48
setGroupCheckable(), méthode 146, 147
setGroupEnabled(), méthode 156
setGroupVisible(), méthode 156
setIcon(), méthode 173
setImageURI(), méthode 35
setIndeterminate(), méthode 116
setIndicator(), méthode 119
setJavaScriptCanOpenWindowsAutomatically(), méthode 170
setJavaScriptEnabled(), méthode 165, 170
setLatestEventInfo(), méthode 339
setListAdapter(), méthode 68
setMax(), méthode 115, 180
setMessage(), méthode 173
setNegativeButton(), méthode 173
setNeutralButton(), méthode 173
setNumericShortcut(), méthode 147
setOnCheckedChanged(), méthode 39
setOnCheckedChangeListener(), méthode 39
setOnClickListener(), méthode 23, 280
setOnItemSelectedListener(), méthode 67, 71
setOrientation(), méthode 46
setProgress(), méthode 115
setProjectionMap(), méthode 269
setQwertyMode(), méthode 147
setResult(), méthode 203
setTables(), méthode 269
setTag(), méthode 91, 97
setText(), méthode 23
setTextSize(), méthode 169
setTitle(), méthode 173
setTypeface(), méthode 160
setup(), méthode 119
setUserAgent(), méthode 170
setView(), méthode 172
setVisible(), méthode 156
setWebViewClient(), méthode 167
setZoom(), méthode 353
SharedPreferences, classe 246, 255
shouldOverrideUrlLoading(), méthode 167
show(), méthode 172
showNext(), méthode 126
SimpleCursorAdapter, classe 270, 300
Singleton, patron de conception 325
size(), méthode 355
SlidingDrawer, widget 130
SOAP, protocole 292
SoftReference, classe 328
Spanned, interface 227
Spinner, widget 70
sp, unité de mesure 380
sqlite3, programme 271
SQLiteDatabase, classe 262
SQLite Manager, extension Firefox 271
SQLiteOpenHelper, classe 262
SQLiteQueryBuilder, classe 267, 309
SSL et HttpClient 296
StackOverflowException, exception 411
startActivityForResult(), méthode 203
startActivity(), méthode 203, 362
startFlipping(), méthode 129
startService(), méthode 324, 332
stopSelf(), méthode 334
stopService(), méthode 332
string-array, élément 238
string, élément 226
strings.xml, fichier 226
supports-screens, élément 381, 388, 389, 397
SystemClock, classe 178

T

TabActivity, classe 118, 207
TabContentFactory(), méthode 121
TabHost
  classe 208
  conteneur 117
  widget 117
TableLayout, conteneur 58, 250
TableRow, conteneur 58
TabSpec, classe 119
TabView, conteneur 207
TabWidget, widget 117
TelephonyManager, classe 362
TextView, widget 33, 76, 85, 160
TextWatcher, interface 78
TimePickerDialog, widget 110
TimePicker, widget 110
Toast, classe 172
toggle(), méthode 38, 132
toggleSatellite(), méthode 353
TrueType, polices 159
Typeface, classe 160

U

unbindService(), méthode 334
unlock(), méthode 132
unregisterContentObserver(), méthode 316
unregisterReceiver(), méthode 200
update(), méthode 266, 311
uptimeMillis(), méthode 178
Uri, classe 196, 297, 362
uses-configuration, élément 404
uses-feature, élément 404
uses-library, élément du manifeste 12
uses-permission
  élément 318
    du manifeste 12
uses-sdk, élément du manifeste 12, 14

V

VerifyError, exception 415
Versions du SDK 14
View
  classe 23
  widget 60, 86, 87
ViewFlipper, conteneur 124
Virus 317

W

WeakReference, classe 328
WebKit, widget 227, 291
WebSettings, classe 169
WebViewClient, classe 167
WebView, widget 163, 398, 399, 400
wrap_content, valeur de remplissage 47
WVGA, résolution d'écran 377

X

XmlPullParser, classe 234
XML-RPC, protocole 292

Le Programmeur

L'art du développement Android 2e édition

À l'aide d'exemples simples et faciles à exécuter, apprenez à développer des applications pour terminaux Android.

Smartphones, PDA et autres terminaux mobiles connaissent aujourd'hui une véritable explosion. Dans ce contexte, Android, le système d'exploitation mobile créé par Google, présente le double avantage d'être gratuit et open-source. Libre donc à tout un chacun d'en exploiter l'énorme potentiel !

Cette seconde édition couvre désormais Android 2. Vous y apprendrez toutes les bases de la programmation sous Android – de la création des interfaces graphiques à l'utilisation de GPS, en passant par l'accès aux services web et bien d'autres choses encore ! Il regorge d'astuces et de conseils qui vous aideront à réaliser vos premières applications.

À travers des dizaines de projets d'exemples, vous assimilerez les points techniques les plus délicats et apprendrez à créer rapidement des applications convaincantes.

Les codes sources des exemples sont téléchargeables sur le site www.pearson.fr.

- Tour d'horizon
- Projets et cibles
- Création d'un squelette d'application
- Utilisation des layouts XML
- Utilisation des widgets de base
- Conteneurs
- Widgets de sélection
- S'amuser avec les listes
- Utiliser de jolis widgets et de beaux conteneurs
- Le framework des méthodes de saisie
- Utilisation des menus
- Polices de caractères
- Intégrer le navigateur de WebKit
- Affichage de messages surgissants
- Utilisation des threads
- Gestion des événements du cycle de vie d'une activité
- Création de filtres d'intentions
- Lancement d'activités et de sous-activités
- Gestion de la rotation
- Utilisation des ressources
- Utilisation des préférences
- Accès et gestion des bases de données locales
- Accès aux fichiers
- Tirer le meilleur parti des bibliothèques Java
- Communiquer *via* Internet
- Utilisation d'un fournisseur de contenu *(content provider)*
- Construction d'un fournisseur de contenu
- Demander et exiger des permissions
- Création d'un service
- Appel d'un service
- Alerter les utilisateurs avec des notifications
- Accès aux services de localisation
- Cartographie avec MapView et MapActivity
- Gestion des appels téléphoniques
- Outils de développement
- Gestion des différentes tailles d'écran
- Gestion des terminaux
- Gestion des changements de plateformes
- Pour aller plus loin

Niveau : Intermédiaire / Avancé

Catégorie : Développement mobile

À propos de l'auteur

Programmeur depuis plus de 25 ans, **Mark Murphy** a travaillé sur des plateformes allant du TRS-80 aux derniers modèles de terminaux mobiles. Il est le rédacteur des rubriques "Building Droids" de AndroidGuys et "Android Angle" de NetworkWorld.

PEARSON

Pearson Education France
47 bis, rue des Vinaigriers 75010 Paris
Tél. : 01 72 74 90 00
Fax : 01 42 05 22 17
www.pearson.fr

Apress

ISBN : 978-2-7440-4159-4

9 782744 041594