

ANDROID

Développer des applications mobiles pour les Google Phones

Florent Garin

Gérant fondateur de la société DocDoku

Certifié Sun Java 2 Programmer et Sun Enterprise Architect for J2EE

Préface de

Sylvain Wallex

Directeur de la technologie de Goojet

Membre de la fondation Apache

DUNOD

Toutes les marques citées dans cet ouvrage sont des marques déposées par leurs propriétaires respectifs.

Illustration de couverture : © Franck Boston - Fotolia.com

<p>Le pictogramme qui figure ci-contre mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du photocopillage.</p> <p>Le Code de la propriété intellectuelle du 1^{er} juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les établissements</p>	<p>d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.</p> <p>Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation de l'auteur, de son éditeur ou du Centre français d'exploitation du droit de copie (CFC, 20, rue des Grands-Augustins, 75006 Paris).</p>
--	--



© Dunod, Paris, 2009
ISBN 978-2-10-054582-7

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2° et 3° a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).

Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

Préface

En vous ouvrant les portes du développement sur Android, ce livre va vous permettre de participer à la révolution qui secoue le monde de la téléphonie mobile depuis l'été 2007.

Avant cette date, la téléphonie mobile suivait une évolution tranquille. Certes, chaque année apportait son lot de mégapixels supplémentaires aux appareils photo intégrés, et le GPS faisait son apparition sur les modèles haut de gamme, mais les nouveaux téléphones n'étaient pas fondamentalement différents de ceux qui les avaient précédés.

Pour l'utilisateur, le téléphone était toujours un appareil aux menus touffus comportant des options de configuration étranges qu'il valait mieux ne pas modifier sous peine de casser le bon fonctionnement de l'appareil. La saisie était assurée par le peu confortable clavier numérique, qui même avec le mécanisme de complétion T9 ne permettait pas de saisir du texte rapidement. Combiné à la limitation de la taille des messages SMS, cela a abouti au fameux « langage SMS » qui est parfois si difficile à déchiffrer par le non initié !

Certains téléphones haut de gamme proposaient une interface tactile, mais elle imposait généralement l'utilisation d'un stylet, parce que ces systèmes étaient la transposition quasi directe sur un écran très petit des principes d'interface homme-machine existants sur les ordinateurs de bureau. Cliquer sur un bouton minuscule ou déplacer une barre de défilement avec un stylet alors qu'on est en train de marcher relève de l'exploit !

Et la très grande majorité des utilisateurs se contentaient des applications de base fournies avec le téléphone, souvent parce qu'ils ignoraient jusqu'à la possibilité d'installer de nouvelles applications, mais aussi parce qu'à part quelques jeux à payer au prix fort, l'offre était relativement limitée et peu intéressante. Le téléphone était fait pour téléphoner, envoyer des SMS et prendre des photos, et rarement beaucoup plus.

Pour le développeur, la situation était complexe : la plupart des téléphones étaient capables d'accueillir des applications JavaME (Java Micro Edition), mais ce standard limité interdisait toute intégration correcte avec le *look and feel* et les fonctions

natives du téléphone, sans parler de l'énorme variété des implémentations et des interprétations du standard qui rendaient le développement d'une application portable digne excessivement difficile. Cela explique aussi pourquoi il y avait peu d'applications disponibles.

En juin 2007, une première révolution eut lieu qui changea radicalement le marché du téléphone mobile : Apple sortit l'iPhone, un téléphone d'un genre entièrement nouveau, doté d'une superbe interface tactile réellement utilisable au doigt, avec des applications simples d'usage mais très efficaces. Dès ce moment, tout nouveau téléphone lui était immédiatement comparé, et chaque constructeur s'est empressé d'annoncer son futur « iPhone killer ».

Il fallut un an avant qu'Apple décide d'ouvrir son téléphone aux applications tierces. Mais développer sur iPhone n'est pas simple : il faut impérativement un Macintosh et coder en Objective-C, langage qui n'est guère utilisé que sur les plateformes Apple. Et il faut ensuite passer la longue et frustrante validation de l'application par Apple, avec des rejets fréquents souvent pour des raisons obscures ou contestables.

En parallèle, les équipes de Google travaillaient sur un projet tout aussi ambitieux. En novembre 2007, ils annoncèrent Android avec leurs partenaires de la Open Handset Alliance : un système d'exploitation pour téléphone mobile moderne, fondé sur Linux, avec une interface tactile similaire à celle de l'iPhone, et intégralement *Open Source*. Il fallut attendre octobre 2008 pour avoir un téléphone Android entre les mains, et c'est clairement une réussite. Android est le seul « *iPhone killer* » crédible.

En rendant son système *Open Source*, et donc gratuit pour les constructeurs, Google jette un pavé dans la mare : les constructeurs du Sud-Est asiatique, réputés pour leur savoir-faire électronique et leurs faibles compétences en logiciel, ont le moyen de produire un téléphone moderne sans devoir payer de royalties pour le système d'exploitation comme c'est le cas avec Windows Mobile. La plupart de ces constructeurs ont annoncé ou ont déjà sorti des modèles fonctionnant avec Android.

Pour le développeur, Android est une bénédiction : le développement se fait en Java, pas un « sous-Java » comme c'est le cas avec JavaME, mais un environnement complet, reprenant une grosse partie de la librairie du JDK et y ajoutant tout ce qui permet de construire très facilement des applications tactiles, graphiques, communicantes, géolocalisées, etc. Et les outils de développement fonctionnent sur votre PC, quel qu'il soit, avec une excellente intégration dans Eclipse.

L'architecture d'Android, comme vous le découvrirez dans ces pages, est d'une grande élégance : en permettant la décomposition d'une application en « activités » communicantes, l'ensemble des applications présentes sur votre téléphone peuvent coopérer, et de nouvelles applications peuvent enrichir celles qui sont déjà présentes. On est très loin du modèle fermé des téléphones qui ont précédé, et même de l'iPhone.

Android ouvre le développement d'applications mobile à l'énorme population des développeurs Java, ce qui ne peut que favoriser l'émergence d'un écosystème très riche. Florent Garin est de ceux-ci, riche d'une grande expérience en Java aussi bien sur le serveur que sur le *desktop*. Il a donc parsemé ces pages de comparaisons utiles avec les librairies et techniques traditionnellement utilisées dans le monde Java.

Cet ouvrage vous ouvrira les portes du développement sur Android. En vous donnant les bases et principes de toutes les fonctions offertes par ce système, il vous permettra de comprendre l'extraordinaire potentiel de cette plate-forme, et d'être rapidement productif pour écrire, je l'espère, la prochaine « *killer app* » à laquelle personne n'avait jamais pensé.

Alors bonne lecture, et à vos claviers !

Sylvain Wallez

Directeur de la technologie de Goojet
Membre de la fondation Apache

Table des matières

Préface	III
Avant-propos	XIII
Chapitre 1 – Présentation	1
1.1 Qu'y a-t-il dans la boîte ?	1
1.2 L'Open Handset Alliance	1
1.2.1 Les constructeurs	2
1.2.2 La licence	3
1.3 Les applications de base	3
1.4 Positionnement par rapport à son environnement	5
1.4.1 L'iPhone	6
1.4.2 Nokia	7
1.4.3 Windows Mobile	7
1.4.4 BlackBerry	8
1.4.5 Palm	9
1.4.6 JavaFx	10
1.4.7 Flash/Flex	10
1.4.8 JavaME	11
1.4.9 Les autres	11

Chapitre 2 – Architecture logicielle	13
2.1 Un Linux Sous le capot	13
2.2 Des bibliothèques C/C++	13
2.3 Un middleware Java	15
2.4 Dalvik	15
2.5 Le JDK	16
Chapitre 3 – Le modèle de programmation	19
3.1 Un développement presque classique	19
3.1.1 Règles de codage	20
3.2 Le SDK Android	22
3.2.1 Plug-in Eclipse	23
3.2.2 Scripts Ant	26
3.3 Développer, Exécuter & Débugger	26
3.3.1 Installation	26
3.3.2 Dalvik Debug Monitor Service (DDMS)	27
3.4 Structure d'un projet	28
3.4.1 Le manifest	30
3.4.2 Les ressources	37
3.5 Et les webapp ?	41
Chapitre 4 – Construire l'interface graphique	45
4.1 Le rendu visuel	45
4.2 Approche programmatique ou déclarative	46
4.3 Les composants graphiques	51
4.3.1 TextView	51
4.3.2 EditText	52
4.3.3 CheckBox	53
4.3.4 ToggleButton	54
4.3.5 RadioGroup	55
4.3.6 Button	56
4.3.7 Spinner	58
4.3.8 AutoCompleteTextView	59

4.3.9	<i>DatePicker</i>	60
4.3.10	<i>TimePicker</i>	61
4.3.11	<i>Gallery</i>	61
4.3.12	<i>ImageView</i>	65
4.3.13	<i>ImageButton</i>	65
4.3.14	<i>ProgressBar</i>	66
4.3.15	<i>AnalogClock</i>	68
4.3.16	<i>DigitalClock</i>	68
4.3.17	<i>RatingBar</i>	69
4.4	Les layouts	70
4.4.1	<i>ListView</i>	70
4.4.2	<i>FrameLayout</i>	71
4.4.3	<i>LinearLayout</i>	71
4.4.4	<i>TableLayout</i>	71
4.4.5	<i>RelativeLayout</i>	72
Chapitre 5	Le modèle de composants	73
5.1	Une forte modularité	73
5.2	Quatre familles de composants	74
5.3	La navigation entre activités	74
5.3.1	L'objet <i>Intent</i>	75
5.3.2	La résolution	76
5.3.3	La communication entre activités	79
5.4	Les services	81
5.4.1	Le langage <i>AIDL</i>	81
5.4.2	Implémentation du service	86
5.4.3	Publication du service	87
5.4.4	Côté client	88
5.5	Le bus de messages	90
5.5.1	Deux modes d'émission	90
5.5.2	Deux façons de s'abonner aux événements	91
5.5.3	Implémentation du receiver	93
5.5.4	Une durée de vie très courte	95

Chapitre 6 – La persistance des données	99
6.1 Sauvegarder l'état des applications	99
6.2 Système de fichiers	100
6.2.1 Lecture d'un fichier	100
6.2.2 Écrire dans un fichier	101
6.3 Les préférences utilisateur	102
6.3.1 Lecture des préférences	103
6.3.2 Écriture des préférences	103
6.3.3 IHM de Configuration	104
6.4 SQLite	106
6.5 Exposer ses données	108
6.5.1 Utiliser les providers	108
6.5.2 Modifier les données du content provider	111
6.5.3 Créer son propre ContentProvider	112
Chapitre 7 – Fonctions IHM poussées	117
7.1 Les menus	117
7.1.1 Les « Option Menus »	118
7.1.2 Les « Context Menus »	120
7.2 Étendre les composants existants	120
7.3 Les animations	121
7.4 Personnalisation en fonction de la configuration	124
7.4.1 Les ressources alternatives	124
7.5 Notifier l'utilisateur	128
7.5.1 Le Toast	128
7.5.2 Barre de statut	130
7.5.3 Les boîtes de dialogue	131
7.6 2D et 3D	134
7.6.1 Graphisme en deux dimensions	134
7.6.2 Graphisme en trois dimensions	137
7.7 App Widgets	138
7.7.1 Définition de l'IHM	139
7.7.2 AppWidgetProvider	140

7.7.3 Écran de configuration	143
7.8 La notion de Task	144
7.9 Styles et thèmes	145
Chapitre 8 – Interaction avec le matériel	149
8.1 Les fonctions de téléphonie	149
8.1.1 Les appels vocaux	149
8.1.2 Réception de SMS	151
8.1.3 Envoi de SMS	152
8.2 Géolocalisation	154
8.2.1 Service de localisation	154
8.2.2 API de cartographie	156
8.2.3 La classe MapView	160
8.3 API réseau bas niveau	164
8.3.1 EDGE et 3G	164
8.3.2 Wi-Fi	165
8.3.3 Bluetooth	166
8.4 L'appareil photo	166
8.5 API Media	168
8.5.1 MediaPlayer	169
8.5.2 MediaRecorder	170
8.5.3 JET Engine	172
8.6 Le vibreur	172
8.7 L'écran tactile	173
8.8 L'accéléromètre	174
Chapitre 9 – Le réseau	177
9.1 Intégration web avec WebKit	177
9.1.1 Approche mixte : web et native	178
9.2 Connexion directe au serveur	178
9.2.1 Les bases de données	178
9.2.2 Quels protocoles réseaux utiliser ?	180
9.2.3 Les web services	181

9.2.4	SOAP	181
9.2.5	POX (<i>Plain Old XML</i>)	184
9.2.6	JSON (<i>JavaScript Object Notation</i>)	184
9.2.7	XMPP	189
Chapitre 10 – Sécurité et déploiement		195
10.1	Signer les applications	195
10.1.1	<i>La phase de développement</i>	195
10.1.2	<i>La phase de packaging</i>	196
10.2	Publier son application	200
10.2.1	<i>Gestion des versions</i>	200
10.2.2	<i>Android Market</i>	201
Annexe		203
Index		211

Avant-propos

Je n'ai jamais vraiment été ce que l'on appelle un « *early adopter* ». Les « early adopters » forment cette catégorie de la population qui se jette systématiquement sur les derniers gadgets technologiques dès leur sortie, essayant au passage les plâtres de produits aux finitions parfois douteuses. Non, je ne fais définitivement pas partie du premier marché des produits électroniques high tech. Par contre, je m'y intéresse fortement, mon métier l'exige d'ailleurs.

Concernant plus spécifiquement le secteur de la téléphonie mobile, j'étais resté plus que dubitatif, il y a près de dix ans, devant l'avènement du WAP. À l'époque, qui coïncidait avec la première bulle Internet, on nous présentait cette technologie comme révolutionnaire, le web mobile était là et il allait changer notre vie. Comme chacun sait, ce fut un fiasco.

Aujourd'hui, les choses ont changé : les téléphones ont des écrans qui se sont agrandis et sont devenus tactiles, ils embarquent désormais des processeurs graphiques permettant une prise en charge de la 3D et de la vidéo, ils sont équipés d'une puce GPS, munis d'un accéléromètre qui se révèle être une interface d'entrée redoutable, et bien sûr, ils sont hautement communicants.

La première fois que j'ai eu un téléphone offrant ce niveau d'équipement, c'était l'iPhone : j'ai senti qu'il s'agissait véritablement d'une révolution. Pour correctement appréhender ce phénomène, il faut bien comprendre que ces « smart phones » ne sont pas de simples ordinateurs en miniature que l'on aurait en permanence sur soi, au fond de sa poche ou dans son sac à main. Non, ils sont bien plus que cela. En effet, grâce à leurs périphériques d'entrées/sorties uniques conjugués à leur statut d'appareil connecté, un champ complètement nouveau d'applications s'ouvre à nous. Des logiciels comme « Around me » qui listent les lieux d'intérêt autour de soi (pharmacies, cinémas, stations service...) ou encore « Shazam » qui permet d'identifier à la volée un morceau de musique simplement en approchant le micro du téléphone près de la source sonore n'auraient jamais pu exister sur un ordinateur classique. La détection des mouvements ou les possibilités tactiles multipoints enrichissent les possibilités offertes aux développeurs des jeux vidéo. L'ajout automatique de

métadonnées de géolocalisation aux photos et aux mails peut être une fonctionnalité capitale pour certains professionnels. Nous n'en sommes qu'au début, les applications de demain, dans le domaine du divertissement ou de l'entreprise, restent encore à inventer.

Si l'iPhone d'Apple semble marquer d'une pierre blanche le début de cette nouvelle ère, la concurrence n'est pas en reste et a bien suivi le mouvement. Nokia a sorti son 5800, BlackBerry le Storm et Palm pourrait bien renaître de ses cendres avec le Pré. Parmi ces alternatives à l'iPhone, il y a une qui sort du lot, à la fois par ses qualités intrinsèques et son mode de développement ouvert. Elle a d'inédit qu'il ne s'agit pas vraiment d'un téléphone mais plutôt d'un système qui s'installe sur du matériel issu de différents constructeurs. Cette particularité pourrait bien faire que ce système s'impose sur le marché des mobiles à l'instar du PC sur le marché des micro-ordinateurs. Ce système, c'est bien sûr Android.

Android est un sujet d'étude très vaste qui mérite un livre à part entière. En effet, son modèle de programmation, son interface graphique, ses composants logiciels, ses fonctionnalités de sauvegarde de données ou encore ses API réseau et de géolocalisation sont uniques.

Ce livre s'adresse à tous ceux désireux de développer des applications professionnelles ou de loisirs fonctionnant sous Android. Il a pour ambition d'accompagner le lecteur du téléchargement du SDK (*Software Development Kit*) au déploiement du programme sur le téléphone. Même s'il est préférable de lire l'ouvrage séquentiellement, il est toutefois possible de le parcourir chapitre par chapitre, une fois que les composants essentiels seront maîtrisés.

Chaque chapitre traite d'un aspect spécifique d'Android accompagné d'exemples de code concrets dont les projets Eclipse sont téléchargeables sur la rubrique « compléments en ligne » du site Internet de Dunod.

1

Présentation

Objectifs

Ce chapitre permet de faire connaissance avec Android. Il décrit l'organisme chargé de son développement, sa licence, son environnement technologique, pour terminer avec les applications proposées par défaut sur les terminaux Android.

1.1 QU'Y A-T-IL DANS LA BOÎTE ?

Définir Android n'est pas chose aisée tant les concepts derrière cet intitulé unique sont nombreux. Pour commencer et en simplifiant à l'extrême, on peut dire qu'Android est un système d'exploitation pour mobile open source développé par la société Google. Tout au long de ce livre, nous découvrirons les nombreuses facettes de ce système et nous élargirons ainsi la définition d'Android.

1.2 L'OPEN HANDSET ALLIANCE

Dès son origine, la démarche de Google a été d'ouvrir le développement d'Android en rassemblant autour de lui et au travers de l'Open Handset Alliance (OHA) un maximum de sociétés. Les membres de ce consortium sont très variés : nous y trouvons des fabricants de téléphones connus tels que Sony Ericsson, Samsung ou Motorola, des opérateurs de téléphonie comme Sprint, T-Mobile ou NTT DoCoMo, des sociétés Internet, Google évidemment mais aussi eBay, des constructeurs de puces électroniques Intel, nVidia, ou encore des acteurs du marché du GPS comme Garmin.

Toutes ces entités se retrouvent donc au sein de cette alliance, pour des raisons qui leur sont propres, pour œuvrer au développement d'Android.

1.2.1 Les constructeurs

Parmi les membres de l'Open Handset Alliance, il y a une catégorie qui intéressera plus particulièrement les consommateurs, c'est celle des constructeurs. C'est en effet dans cette liste qu'il faudra chercher son futur combiné téléphonique. Il est donc crucial pour le succès de la plateforme que cette liste soit bien fournie avec de préférence des poids lourds du secteur. Aujourd'hui, nous pouvons recenser :

- HTC
- Motorola
- Samsung
- Sony Ericsson
- Toshiba
- LG
- Huawei
- Asus

Cependant, en 2009, le choix de terminaux était encore très limité. Seul le Taïwanais HTC avait véritablement dégainé en proposant le Dream (G1) dès fin 2008 puis son évolution le Magic (G2).



Figure 1.1 – G1 HTC

Depuis le premier trimestre 2009, les choses semblent s'accélérer fortement. Les constructeurs ont fait de nombreuses annonces, Nous devrions donc finir l'année avec une dizaine de terminaux sous Android. Toutefois, à l'instar de Samsung qui compte maintenir une gamme de téléphones motorisés tour à tour par Windows Mobile, Symbian, leur propre OS et Android, la plupart des constructeurs demeurent prudents et ne lâchent pas leurs anciens systèmes d'exploitation.

En outre, il est intéressant de noter l'initiative d'Archos qui travaille sur une tablette Internet, évidemment tactile, sous Android. Asus a aussi annoncé au Mobile World Congress sa volonté de proposer Android sur ses netbooks. Manifestement la plateforme se sent à l'étroit sur les téléphones portables, on peut probablement anticiper que les prérogatives d'Android pourraient s'étendre à tout type d'appareil remplissant des fonctions multimédias et ayant des capacités de communication importante.

1.2.2 La licence

Comme il est mentionné sur le site officiel, la licence préférée d'Android est l'« Apache open source licence v2 ». Le terme « préférée » signifie que le projet Android pourra dans une certaine mesure accepter et intégrer du code source n'ayant pas été publié sous cette licence. Néanmoins, de façon prévisible, seules les licences « open source » approuvées par l'Open Source Initiative - organisation dont l'objet est de labéliser les licences « open source » - sont acceptées. Les contributeurs directs au projet, individus ou sociétés, devront, par contre, signer (il est possible de le faire en ligne) un agrément de licence.

La licence Apache a été choisie pour son côté « business friendly » comme on dit. Plus clairement, la licence Apache, contrairement aux licences GPL, autorise la reprise de tout ou partie du code dans un produit sous une autre licence y compris propriétaire.

1.3 LES APPLICATIONS DE BASE

Les terminaux Android de base sont livrés avec un ensemble d'applications dénommées « *Core Applications* ». En fonction des constructeurs, d'autres logiciels pourront être préinstallés sur le téléphone mais les Core Applications constituent un socle commun minimum obligatoire.

Comme Android est destiné à motoriser en premier lieu des téléphones, parmi ces applications se trouvent tout logiquement le composeur de numéro et la gestion des contacts :



Figure 1.2 — Composeur

Ensuite viennent des applications plus sophistiquées comme Google Maps.



Figure 1.3 — Application de cartographie

Couplée au GPS si l'appareil en est doté (attention, les spécifications d'Android n'imposent pas que tous les téléphones soient équipés d'un système de géolocalisation), l'application Maps prend tout son sens et montre bien de quoi la plateforme est capable.

Autre application phare, le navigateur bien sûr.



Figure 1.4 — Un navigateur très performant

Il s'agit d'un vrai navigateur qui n'a rien à voir avec les butineurs wap d'il y a quelques années. Motorisé par WebKit (le cœur de Safari, le navigateur du Mac), ce navigateur est en mesure d'interpréter l'HTML, le CSS, le JavaScript dans leur dernière version, au même titre que ceux pour les ordinateurs de bureau.

Grâce à un système de zoom, il est possible de consulter tous les sites. Néanmoins pour une utilisation agréable, il vaut mieux qu'une version mobile du site, à l'ergonomie adaptée, soit disponible.

1.4 POSITIONNEMENT PAR RAPPORT À SON ENVIRONNEMENT

Android arrive sur un marché déjà bien encombré. La concurrence y est féroce, Nokia domine le secteur avec un peu plus d'un tiers de parts de marché. Toutefois Apple, en faisant une entrée fracassante, a prouvé que les positions étaient loin d'être figées et qu'un nouveau venu avait sa chance à condition bien sûr d'innover. D'autre part, l'approche ouverte et multiconstructeurs d'Android conjuguée à la touche Google laisse espérer un avenir radieux à la plateforme.

1.4.1 L'iPhone

L'iPhone, sans aucune contestation possible, est un téléphone qui fera date dans l'histoire de cette industrie. Apple a fixé en partie certains standards pour les téléphones de demain : le large écran tactile, de préférence multipoints, paraît incontournable, l'accéléromètre aussi. L'esthétique de l'interface graphique, la fluidité des animations, les zooms sur les images ou les cartes faits avec deux doigts posés sur l'écran ont donné brutalement un sacré coup de vieux à la concurrence.

Il semble aussi certain aujourd'hui qu'un smartphone se doit de disposer de sa boutique en ligne ou « App Store ». Pour Apple, cette dernière est un beau succès : début 2009, il n'y avait pas moins de 5 millions de téléchargements par jour. Un écosystème s'est ainsi créé et des milliers de développeurs amateurs ou professionnels tentent leur chance en proposant leurs créations.

Malgré toutes les qualités de l'iPhone, celui-ci est loin d'être exempt de défauts. Au-delà des classiques reproches qui lui sont faits sur l'absence de la fonction « couper/coller »¹ ou sur le peu de profils Bluetooth supportés (pas de transfert de fichiers, pas d'oreillette stéréo...), au bout du compte, le problème numéro un de l'iPhone est son aspect fermé !

Pas de doute là-dessus, l'iPhone est bien un produit de la firme à la pomme. Au moment de la rédaction de ce livre, le SDK (*Software Development Kit*), certes gratuit, n'était disponible que pour les seuls possesseurs de Mac, pas de version Windows ni Linux. Pour publier ses applications, le développeur a le choix entre le « Standard Program » à 99 \$ et l'« Enterprise Program » à 299 \$. Dans le premier cas, la distribution se fera au travers de l'App Store accessible publiquement par iTunes depuis un ordinateur (PC ou Mac) ou directement depuis le téléphone. Dans le second cas, le programme sera déployé depuis les serveurs internes de l'entreprise. Ce qui permet de distribuer les logiciels propriétaires métiers.

Ce qui est gênant dans le système Apple, ce n'est pas tant le prix de 99 \$, après tout les coûts liés à l'hébergement et l'exploitation de la boutique en ligne sont bien réels. Le problème vient de la position centrale et incontournable d'Apple. Contrairement à Android (voir chapitre 10, « Sécurité et déploiement »), la distribution d'applications pour l'iPhone ne peut se faire que par iTunes, chasse gardée d'Apple. Ce dernier se réserve donc le droit d'accepter ou de refuser, sans fournir aucune explication, les logiciels dans son App Store. Le logiciel NetShare en a fait les frais, tout d'abord autorisé à intégrer l'App Store puis retiré. Ce logiciel est un proxy SOCKS grâce auquel on peut se servir de son iPhone comme d'un modem concurrençant au passage les clés 3G des opérateurs ! C'est probablement la raison de son retrait.

1. Le tout dernier firmware 3.0 de l'iPhone corrige en partie les défauts de jeunesse du téléphone, en offrant le couper/coller et la fonction modem. Cependant il ne faut pas se réjouir trop vite le partage de la connexion 3G avec un ordinateur ne peut se faire qu'avec le consentement de l'opérateur, c'est-à-dire après s'être acquitté d'un abonnement supplémentaire.

Cette politique unilatérale d'Apple fait couler beaucoup d'encre. Les plus gentils disent que c'est pour garantir une certaine qualité et s'assurer que les programmes en question ne sont pas des malwares, d'autres crient purement et simplement à la censure en arguant qu'Apple ne fait que défendre ses intérêts commerciaux et ceux de ses partenaires en se réservant les applications les plus lucratives.

1.4.2 Nokia

Nokia est le plus grand constructeur de téléphonie du monde devant Motorola. Sa gamme est très large et couvre tous les segments : du premier prix vendu 1 € avec un abonnement auprès d'un opérateur jusqu'au mobile PDA avec GPS et enregistreur vidéo. Les Nokia sont généralement réputés pour leur grande fiabilité. Leur système d'exploitation est le Symbian OS développé par la société éponyme aujourd'hui propriété exclusive de Nokia. Bien que l'OS Symbian se retrouve sur quelques appareils Samsung ou Sony Ericsson, ce système d'exploitation est fortement connoté Nokia, ce qui est un frein à son adoption par d'autres constructeurs.

La marque finlandaise l'a bien compris et opère aujourd'hui une stratégie similaire à Android. En effet, Symbian devrait passer en open source (sous Eclipse Public License) et son évolution devrait se faire sous l'égide de la Symbian Foundation. Le but de Nokia est de s'ouvrir, d'accélérer le développement de son système tout en créant une communauté. Par rapport à l'Open Handset Alliance, la Symbian Foundation a un peu de retard, l'initiative est venue après, et le consortium monté par Nokia comporte donc moins de membres. Enfin, rien n'est joué : certains industriels sont d'ailleurs présents dans les deux associations. Tout dépendra de la capacité des leaders, Google et Nokia, à mobiliser véritablement leurs partenaires.

1.4.3 Windows Mobile

Windows Mobile, WiMo pour les intimes, est l'OS mobile de Microsoft. C'est une évolution de Windows Pocket PC, ancêtre de Windows CE. Cet OS, sans avoir jamais déchaîné les passions, a réussi au fil des années à s'octroyer une part de marché honorable.

Son intérêt a sans doute été suscité par son affiliation à la famille d'OS Windows, ultra-dominante sur le bureau. Un autre avantage souvent cité est la facilité de développement apportée grâce à l'environnement cliquodrome de Visual Studio qui a su faire venir au développement mobile les développeurs VB.

Aujourd'hui, Microsoft met en avant, entre autres, son intégration avec Exchange qui, par le biais du protocole Exchange ActiveSync, offre la fameuse fonctionnalité de push mail qui, comme son nom l'indique, pousse les nouveaux messages du serveur de mail vers le téléphone.

1.4.4 BlackBerry

Tout comme l'iPhone, le BlackBerry est aussi un téléphone très en vue. Au départ clairement positionné sur le marché des entreprises, la fonction majeure qui a fait décoller le BlackBerry était le push mail. L'utilisateur n'a alors plus besoin de consulter périodiquement sa boîte pour vérifier s'il n'a pas de nouveaux messages. Ceux-ci lui parviennent directement comme un banal SMS. Cette fonctionnalité est assurée par les serveurs d'infrastructure du fabricant RIM (Research In Motion) avec un protocole propriétaire. Le mail est donc le point fort des BlackBerry qui faisaient rêver de nombreux cadres et dirigeants, même Barack Obama en était paraît-il accroc !

Aujourd'hui, les lignes bougent, l'iPhone s'est rapproché du monde de l'entreprise en proposant aussi une fonctionnalité de push mail avec le protocole d'Exchange ActiveSync et BlackBerry, de son côté, fait le chemin inverse en se dotant de capacité multimédia et d'une ergonomie tactile avec le BlackBerry Storm. RIM se rapproche ainsi du grand public ou plutôt veille à ce que sa clientèle d'entreprise ne soit pas trop attirée par l'élégance de l'iPhone.

Android, quant à lui, n'étant pas porté par un unique constructeur, ne cible pas spécifiquement les particuliers ou le monde professionnel. La plateforme est généraliste, on peut y développer aussi bien des jeux que des applications métiers.



Figure 1.5 — BlackBerry 8800

1.4.5 Palm

Après avoir connu son heure de gloire en étant l'une des premières sociétés à commercialiser des assistants numériques, Palm était depuis sur le déclin. Il y a quelques années, Palm a même cédé aux sirènes de Windows Mobile en proposant certains de ses appareils sous l'OS de Microsoft. Palm avait cessé d'innover et se devait de réagir face aux assauts d'Apple et de Google.

La réponse s'appelle Palm Pré. Ce téléphone sera le premier qui tournera sous le nouvel OS de Palm, le WebOS. WebOS a été dévoilé à l'occasion du CES 2009 de Las Vegas. Et il faut reconnaître que l'accueil de l'assistance fut plus que chaleureux. On a même pu assister à un certain buzz dans les jours qui ont suivi. L'intérêt que suscite à nouveau Palm est mérité. Le Palm Pré a un look et une ergonomie très attractifs. Visuellement le WebOS est plutôt réussi, il y a même eu une polémique sur sa ressemblance, il y est vrai assez poussée, avec l'iPhone. Le fait que certains hauts dirigeants de Palm sortent tout droit de l'entreprise de Cupertino ne tend pas à calmer les esprits.



Figure 1.6 — Le Palm Pré et son look arrondi

(Photo de whatleydude (Flickr) sous licence Creative Commons 2.0)

Quoi qu'il en soit, si Palm s'est plus ou moins inspiré de la concurrence (ce qui est en fin de compte dans l'ordre des choses), il a eu la bonne idée d'y adjoindre quelques améliorations : une batterie rechargeable, un chargeur sans fil, un clavier amovible ou encore le Bluetooth. L'OS lui-même à l'air aussi engageant : il supporte le multitâche, un lecteur flash, une intégration web omniprésente et surtout Palm insiste particulièrement sur le framework de développement (Mojo SDK) qui s'appuie sur JavaScript, JSON, HTML, XML et CSS, en somme que des technologies « web ».

Cette approche est séduisante mais de là à dire que tout développeur d'application web pourra coder pour le Pré sans effort est un raccourci un peu rapide. Il faudra évidemment se familiariser avec l'API. De plus, de nombreuses interrogations demeurent : comment se fait l'accès aux couches basses du matériel, que vaut l'IDE ? Quelle sera la politique de l'App Store ?

Bref, ce téléphone est assurément prometteur et pourrait être un concurrent sérieux des combinés Android sur le haut de gamme, mais il faudra attendre sa sortie pour se faire un avis définitif.

1.4.6 JavaFx

JavaFx est une technologie développée par Sun Microsystems pour construire des interfaces graphiques à l'aide d'un langage de script. JavaFx vient aussi avec un environnement d'exécution doté d'API d'animation vectorielle, d'une collection de widgets, de divers codecs audio et vidéo.

JavaFx cible le marché des RIA (*Rich Internet Applications*), ses concurrents sont donc davantage le trio Flash/Flex/Air d'Adobe ou Silverlight de Microsoft qu'Android. Cependant, Sun ambitionne de fournir un environnement JavaFx sur un maximum de matériel : du PC de bureau au téléphone mobile en passant par les téléviseurs... JavaFx est donc une option sérieuse à considérer pour développer une application mobile.

La force de JavaFx Mobile est de pouvoir fonctionner sur une simple base JavaME. L'intérêt de ce tour de force est évident : le parc de téléphones compatibles JavaME étant conséquent, les applications JavaFx se retrouvent *de facto* déployables sur un nombre impressionnant de mobiles.

Contrairement à Android qui est une plateforme centrée sur le mobile, JavaFx, par son approche universelle, offre la possibilité de déployer le même code sur le poste de travail ou sur le téléphone. Cet avantage doit toutefois être abordé avec mesure ; d'un côté on ne conçoit pas un programme pour téléphone comme n'importe quelle application, surtout sur le plan de l'ergonomie. D'un autre côté, cela pourrait attirer à JavaFx une certaine clientèle.

Depuis le rachat de Sun par Oracle, des doutes planent quant à l'avenir de JavaFx. Les RIA ne font pas vraiment partie des secteurs traditionnels d'Oracle qui, en général, se concentre sur le marché des systèmes d'entreprises (ERP, bases de données, serveurs d'applications...).

1.4.7 Flash/Flex

Flash est une technologie développée par Adobe qui existe depuis plus de dix ans sur les ordinateurs de bureau et qui permet d'insérer du contenu dynamique (animation, film...) dans une page web. Flash a su s'imposer pour de nombreuses raisons parmi lesquelles on peut citer sa faible empreinte mémoire et CPU, sa capacité à se lancer très vite, l'esthétisme des applications dû au fait que les outils de production sont des logiciels destinés aux designers ou encore son installation si aisée que l'utilisateur non averti se rend à peine compte qu'un téléchargement suivi d'un déploiement vient de se produire sur son poste...

Fort de ce succès et de l'ubiquité de Flash, Adobe a bâti une surcouche, Flex, destinée à faciliter le développement de véritables applications ; celle-ci élargit le domaine de compétence de Flash jusqu'ici cantonné à servir du contenu multimédia.

Aujourd'hui, Adobe travaille à porter Flash sur un maximum d'appareils ; il collabore d'ailleurs avec Google pour cela. Flash n'est donc pas un concurrent d'Android, ils ne sont pas à mettre sur le même plan. Cependant, on imagine mal Adobe s'arrêter en si bon chemin et ne pas pousser Flex jusque sur le mobile pour devenir une plateforme de développement mobile.

1.4.8 JavaME

JavaME (*Java Micro Edition*) est l'équivalent du JavaSE (*Java Standard Edition*) ciblant les appareils mobiles, téléphones, carte à puces, TV... La plateforme se targue d'un déploiement impressionnant se chiffrant à plusieurs milliards d'unités.

Avec l'apparition récente de JavaFx, on peut se poser la question de l'avenir de JavaME. En réalité JavaFx est une surcouche graphique au-dessus de JavaME. Peu à peu JavaME pourrait donc s'effacer devant JavaFx pour ne devenir qu'une discrète interface gérant les interactions avec l'OS et le matériel. Ceci sera surtout vrai pour les téléphones haut de gamme.

Sur le segment des premiers prix, JavaME a toutes ses chances de rester la plateforme de prédilection. Son omniprésence sur ce type de mobile en fera un concurrent coriace pour Android. Mais ce n'est sans doute pas la cible de l'OS de Google.

1.4.9 Les autres

Le marché plus traditionnel des PC, après avoir connu plusieurs années de hausse ininterrompue est aujourd'hui en train de péricliter sérieusement. Inutile donc de dire que le secteur de la mobilité qui lui est dynamique, attise les convoitises.

Les entreprises éditrices de distributions Linux y voient là un puissant relais de croissance. La société sud-africaine Canonical Ltd qui édite le système d'exploitation Ubuntu, a dévoilé récemment Ubuntu Mobile. Ce projet est soutenu par Intel et bénéficiera certainement du support actif de la communauté open source Ubuntu.

D'autres sociétés proposent également des environnements mobiles à base de Linux. Par exemple, la société japonaise Access, qui a racheté PalmSource, l'éditeur de Palm OS (l'ancien OS de Palm, pas le nouveau !), développe *Access Linux Platform* (ALP).

Ce qu'il faut retenir

Comparer les offres concurrentes à celle d'Android est compliqué tant les stratégies et les approches sont différentes ; certains proposent un système intégré (téléphone + OS), d'autres tentent d'unifier le modèle d'exécution des applications entre les ordinateurs de bureau et les mobiles.

Android quant à lui se positionne comme un système d'exploitation et un environnement de développement open source, dédié aux appareils mobiles, et indépendant de toute plateforme matérielle spécifique.

2

Architecture logicielle

Objectifs

Après avoir examiné Android sous l'angle de l'utilisateur, ce chapitre met le focus sur les aspects techniques internes de la plateforme.

Son OS, son environnement d'exécution d'applications, le processus de développement y sont abordés.

2.1 UN LINUX SOUS LE CAPOT

Au-delà du framework de développement, Android est une pile logicielle qui repose sur le couple, en passe de devenir un classique : Linux/Java. Le noyau Linux employé est le 2.6 dont on peut remarquer au passage qu'il constitue un composant central d'Android qui n'est pas publié sous licence Apache 2.0 mais sous GPL v2. Comme n'importe quel OS, ce noyau a la responsabilité de gérer le matériel à l'aide de drivers, la mémoire, les processus ou encore les couches réseaux basses.

2.2 DES BIBLIOTHÈQUES C/C++

Au-dessus du noyau proprement dit se loge un ensemble de bibliothèques C/C++ constituant les couches bases du système. Parmi celles-ci on peut noter l'incontournable lib (bibliothèque système C standard) dont l'implémentation a été adaptée pour l'occasion à un mode embarqué.

Il y a également diverses bibliothèques graphiques assurant le rendu vectoriel des polices de caractères ou encore le rendu 2D et 3D. Android dispose d'ailleurs d'une implémentation de la spécification OpenGL ES 1.0 (ES signifiant *Embedded Systems*) qui s'appuie sur l'accélération 3D matériel si le téléphone en est pourvu ou effectue un rendu purement « *software* » dans le cas contraire. Il est intéressant de voir qu'OpenGL ES tend à devenir la norme en matière d'API 3D pour le mobile. L'iPhone et Nokia (Symbian OS) ont retenu la même API. Évidemment, Microsoft lui préfère la version mobile de Direct3D, le concurrent de toujours d'OpenGL.

Côté « média », Android ne manque de rien : il lie les formats d'images PNG et JPG, il est capable d'encoder et de décoder des flux audio et vidéo aux formats AAC, H.264, MP3, MPEG-4 ou Ogg Vorbis.

Enfin, pour clore le tableau, même s'il ne s'agit pas tout à fait de librairies bas niveau, on peut ajouter WebKit et SQLite.

WebKit est le moteur du navigateur web qui peut également être manipulé par du code dans les applications. Ce moteur a le vent en poupe et se retrouve aujourd'hui incorporé dans de nombreux navigateurs : Safari sur Mac, Konqueror (KDE Linux), Google Chrome, et surtout dans bon nombre de téléphones portables (Nokia, iPhone, Palm Pré...).

Mozilla Firefox tourne avec le moteur Gecko. En environnement embarqué, WebKit est souvent préféré à Gecko car il est moins gourmand en ressource que ce dernier, qui est capable de lire non seulement du contenu web tel que l'HTML, le CSS et le JavaScript, mais aussi le format XUL (langage d'interface utilisateur propre à Mozilla), base des extensions Firefox.

WebKit est donc léger car centré uniquement sur les technologies web standard qu'il implémente fort bien ; c'est un des rares moteurs à obtenir le score de 100/100 au test de compatibilité des navigateurs Acid3.

SQLite est un moteur de base de données sous forme de librairie C et par conséquent destiné exclusivement à être encapsulé dans un logiciel. SQLite se trouve ainsi utilisé comme système de stockage interne dans énormément d'applications, c'est même la référence des moteurs SQL embarqués.

Notre système Android est donc constitué d'un noyau Linux et d'une suite de bibliothèques C/C++ fort utile. Mais comment se fait l'accès à ces librairies ? Pour l'instant exclusivement par l'API Java d'Android. Il n'est donc pas possible de faire des appels directs aux couches basses et il faudra se contenter des fonctionnalités exposées.

2.3 UN MIDDLEWARE JAVA

Si nous remontons les couches de la plateforme, après le noyau Linux et les bibliothèques C/C++, nous arrivons au middleware, l'environnement d'exécution Android. Cet environnement est constitué en premier lieu d'une machine virtuelle qui ne peut prétendre à l'appellation Java car elle ne respecte pas les spécifications officielles que l'on peut trouver à l'adresse : http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html

Cette machine virtuelle a été dénommée Dalvik. Le chapitre suivant passe en revue le fonctionnement de cette VM. Dalvik interprète donc du code, dont les sources sont écrites en Java. À l'instar des véritables Java Virtual Machine, Dalvik vient avec son ensemble de core API qui correspond aux *Java standard libraries*.

2.4 DALVIK

Google a fait le choix de s'écarter des standards Java et de proposer sa propre machine virtuelle. Fondamentalement, la démarche est la même que celle du framework GWT (*Google Web Toolkit*), Google ne garde de Java que le langage et une partie des API standard et met de côté la plateforme d'exécution (JVM) ; dans le cas de GWT, la cible sur laquelle s'exécute le code est le navigateur et son interpréteur JavaScript, et pour Android il s'agit de Dalvik.

La raison d'être de Dalvik est de fournir un environnement optimisé aux petits objets pour le mobile où les ressources CPU et mémoires sont précieuses. Par exemple, pour de meilleures performances sur un système embarqué, Dalvik est une machine virtuelle dite « *registered-based* », basée sur les registres, et non pas « *stack-based* » comme la JVM de Sun.

Par ailleurs, sur Android, par défaut, chaque application tourne dans son propre processus Linux, c'est-à-dire sa propre VM. L'un des points forts d'Android étant ses capacités multitâches, il doit être possible de lancer efficacement plusieurs VM Dalvik sur le terminal.

À chaque fois qu'une application est démarrée, une VM se crée, puis le processus meurt à la fermeture du programme. On est bien loin du modèle de la JVM pensée pour être rarement arrêtée ou relancée et censée exécuter simultanément plusieurs applications distinctes isolées entre elles par une hiérarchie de *ClassLoader*.

Les besoins d'Android imposaient vraiment de concevoir une machine virtuelle spécifique.

2.5 LE JDK

Les applications Android ne s'exécutent pas sur une JVM standard certes, mais avec quel JDK faut-il coder et compiler les applications ?

Réponse : le JDK 1.5 et 1.6, comme d'habitude en somme a-t-on envie de dire. Par contre, il ne faut pas se méprendre, l'ensemble des classes utilisables dans un environnement Android n'est qu'un sous-ensemble du JDK. Une fois de plus, comme GWT, seule une portion de l'API standard a été portée vers l'environnement d'exécution.

Donc, pour bien comprendre, cela veut dire que dans la pratique, un développeur pourrait penser bien faire en codant son application Android en utilisant toute la palette des classes du JDK comme les API JAXB (package `javax.xml.bind.*`) ou Java Swing (package `javax.swing.*`). Mais, ce développeur aurait une mauvaise surprise en tentant de compiler son application, car il n'y parviendrait pas !

En effet, les applications Android se compilent en utilisant les options de Cross-Compilation de `javac` « `-bootclasspath android.jar -extdirs ""` » qui ont pour double effet de remplacer la traditionnelle librairie `rt.jar` qui contient les classes Java de base et de retirer du classpath les jar d'extensions placés dans le répertoire `jre/lib/ext` du JDK.

Heureusement, il est possible de configurer la plupart des environnements de développement de manière à changer le bootclasspath, ainsi l'IDE prémunira l'utilisateur d'importer des classes auxquelles il n'a pas droit.

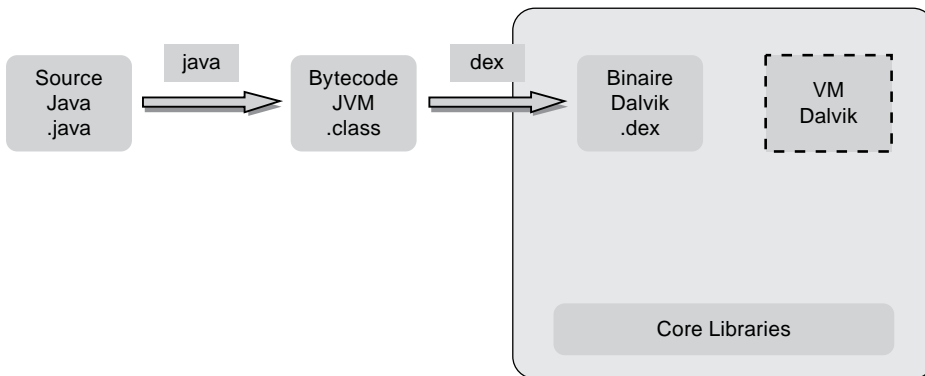


Figure 2.1 — La chaîne de compilation d'Android

L'assistant de création de projet Android du *plug-in* Eclipse génère des fichiers de configuration de projet qui vont dans ce sens.

Une fois les fichiers `.class` créés avec les outils du JDK standard, ce sont les outils Android qui prennent le relais notamment la commande « `dx` » qui transforme ces mêmes `.class` en fichier au format `dex` propre à la machine virtuelle Dalvik.

Ce qu'il faut retenir

Avec Android, Google réitère ce qu'il avait fait pour son kit de développement web AJAX GWT : il prend de Java ce qui l'intéresse et laisse de côté le reste !

Le langage Java et sa syntaxe sont conservés tels quels, son API standard n'est pas modifié non plus, du moins le sous-ensemble qui a été retenu, car une partie de cette API n'a pas été portée vers Android.

Enfin, le format binaire des classes est carrément incompatible, la machine virtuelle Android ne respecte pas la spécification de Sun de la JVM.

Le « *Write once, run anywhere* » n'est plus de mise, c'est pour cela que Dalvik ne peut se prévaloir du qualificatif de JVM.

3

Le modèle de programmation

Objectifs

Après les chapitres précédents qui présentaient les entrailles d'Android, ce chapitre passe à la pratique en étudiant concrètement comment un développement Android se déroule. Si le développeur Java traditionnel ne sera pas dépaycé, il n'en reste pas moins qu'au-delà des simples impressions d'être en terrain connu, un développement Android nécessite de bien maîtriser les ressorts et les concepts de la plateforme.

3.1 UN DÉVELOPPEMENT PRESQUE CLASSIQUE

Il ne faut pas se tromper, on ne code pas une application Android comme on coderait un programme Java standard destiné à être exécuté sur un poste de travail. Il convient de se méfier des raccourcis : le fait d'utiliser Java et Eclipse ne signifie pas qu'on ne doit pas adapter ses pratiques de développement¹.

Sur un mobile, la capacité mémoire est limitée, la puissance CPU forcément plus réduite que celle du poste de développement. De plus, un téléphone portable possède une batterie qu'il s'agit de ménager ! Android est un environnement embarqué, certaines règles de codage s'imposent donc.

1. À ce sujet, voir aussi DiMarzio, Jerome (J.F.), *Android, A programmer's guide*, McGraw-Hill, USA 2008.

3.1.1 Règles de codage

Si une application Android est une application embarquée, elle ne peut tout de même pas se classer dans la catégorie des applications temps réel critique. La plateforme intègre donc un *Garbage Collector* (GC), les objets créés et déréférencés seront bien éliminés de la mémoire. Néanmoins, ce qui est valide pour un programme Java classique est d'autant plus vrai sur environnement contraint. À chaque passage du GC, la machine virtuelle a tendance à marquer un temps d'arrêt. Au niveau de l'interface graphique, cela génère une impression négative, gâchant l'expérience utilisateur.

Il est donc préférable de minimiser la création d'objets temporaires. Attention, toutefois à ne pas tomber dans l'excès inverse. Refuser catégoriquement de créer dynamiquement des objets en allouant statiquement la mémoire ou générer tous les objets au travers d'un pool conduirait à sacrifier le design de l'application qui deviendrait difficilement maintenable.

Pour améliorer les performances, il est recommandé également de déclarer statiques les méthodes pouvant l'être, c'est-à-dire celles ne modifiant pas les attributs de la classe, et de bien tagger les constantes avec le modificateur final.

Il est aussi plus performant d'accéder directement aux attributs des classes. Pour ne pas casser l'encapsulation des objets, cette recommandation ne devra s'appliquer qu'aux attributs internes, les appels aux getters et setters sont donc à proscrire à l'intérieur même de la classe. Un accès direct à une variable membre est donc plus rapide que de passer par une méthode mais un accès par une variable locale est encore plus performant. Par conséquent, si dans une méthode un attribut de la classe est accédé à maintes reprises, il peut être opportun d'affecter l'attribut à une variable locale :

```
int nbreElements = this.nbreElements;
Object[] elements = this.elements;
for (int i = 0; i < nbreElements; i++)
    System.out.println(elements[i]);
```

De la même façon, il est plus performant d'éviter les appels de méthodes au travers d'interface. Par exemple, la « API Java Collections » définit les interfaces *Map*, *Set*, *List*, *SortedSet*, *SortedSet*... Plusieurs implémentations de ces interfaces sont disponibles : pour l'interface *List*, il y a, entre autres, les classes *ArrayList* et *LinkedList*.

Chacune de ces implémentations fournissant donc la même API, elle offre des niveaux de performance variables selon les usages (lecture seule, modification fréquente...). Les tutoriaux officiels de Sun préconisent par conséquent de déclarer les objets avec leur interface afin de s'autoriser à changer le choix de l'implémentation si besoin est. Il faudrait alors écrire :

```
List<Object> maList = new ArrayList<Object>();
```

Et non :

```
ArrayList<Object> maList = new ArrayList<Object>();
```

Sur Android, il est préférable de faire le contraire ! Bien sûr, ce principe doit aussi être appliqué avec parcimonie. Si le code en question constitue une API publique, la neutralité apportée par les interfaces vaut probablement la petite perte de performance.

Depuis la version 5 (JDK 1.5) de Java, un nouveau mot-clé *enum* a été introduit au langage. Ce mot-clé sert à définir les types énumérés remplaçant avantageusement les listes de constantes.

Avant l'ajout de ce mot-clé, la notion de type énuméré était implémentée comme ceci :

```
public final static int BANANE = 1;
public final static int POMME = 2;
public final static int ORANGE = 3;
```

Avec l'enum, cela donne :

```
public enum Fruit{BANANE, POMME, ORANGE}
```

Cette dernière forme est plus robuste que la précédente car elle apporte la garantie à la compilation qu'aucune valeur autre que celles prévues ne pourra être employée pour un type énuméré donné. Les énumérés étant aussi des classes, ils possèdent certaines méthodes utiles à leurs manipulations : *valueOf(arg)*, *values()*.

Le revers de la médaille est qu'utiliser un énuméré est plus coûteux en mémoire et en cycles CPU que des constantes. Sur une plateforme limitée comme un téléphone, ce surcoût peut être significatif. Comme toujours, dans le cadre de la définition d'une API publique, on pourra par contre privilégier la robustesse et la clarté du code à sa performance.

Plus généralement, il est souhaitable de bien avoir conscience des instructions VM et même processeur qui se cachent derrière chaque fragment de code source. L'usage d'une construction telle que « for each » sur un objet *Iterable* génère par le compilateur l'appel à la méthode *iterator()* puis des méthodes *hasNext()* et *next()* à chaque boucle.

Pour parcourir une *ArrayList* qui utilise comme structure de données interne un tableau, il est plus efficace de le faire directement avec la méthode *get(index)*.

Il y a d'autres cas analogues. Par exemple, l'accès à une variable membre ou à une méthode privée depuis une classe interne non statique ne se fait en réalité qu'à travers des méthodes dites « *synthetic methods* », générées par le compilateur afin d'atteindre ces variables et méthodes aux visibilité réduites. Si on augmente la visibilité de ces éléments, ils deviendront accessibles depuis la classe interne et il ne sera plus nécessaire de passer par des méthodes intermédiaires autogénérées : on gagnera ainsi en efficacité.

Il n'est pas possible de lister toutes les subtilités d'optimisation possibles, il faut juste ne pas perdre à l'esprit que la cible de l'application n'est pas une machine dual core avec 2 Go de RAM mais un mobile aux capacités nettement plus réduites. En cas de doute sur un algorithme il vaut mieux se documenter et analyser les opérations qui seront concrètement exécutées.

3.2 LE SDK ANDROID

Le kit de développement (Software Development Kit) Android se présente sous la forme d'un fichier zip qu'il suffit de décompresser dans un répertoire quelconque.

Au début du deuxième semestre 2009, la dernière version stable de ce kit était la 1.5 r2.

Une fois le kit installé, l'utilisateur se retrouve avec l'arborescence suivante :

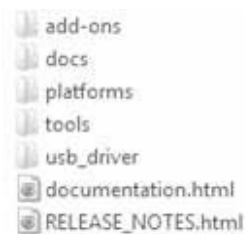


Figure 3.1 — Le contenu du SDK

Le répertoire « docs » contient la documentation HTML d'Android. Il s'agit exactement du site <http://developer.android.com>. Si l'on souhaite bénéficier d'une version mise à jour, il vaut mieux sans doute visiter le site web.

Cependant pour la javadoc qui est une rubrique de cette documentation, il est toujours intéressant d'avoir une version en local sur son ordinateur en cas de défaillance réseau.

Le répertoire « samples » (sous-répertoire de docs) regroupe quelques exemples de code dont il est fait référence par endroits dans la documentation.

Le répertoire « tools » liste les utilitaires Android. La plupart de ces outils sont à utiliser en ligne de commande. Ces commandes sont suffisamment nombreuses pour mériter qu'un paragraphe dans ce même chapitre leur soit réservé.

« usb_driver » stocke les drivers Windows (pour les systèmes x86 et amd64) nécessaires pour déboguer les applications directement sur le téléphone et non plus sur l'émulateur. Le chemin de ces drivers est à spécifier à Windows par la boîte de dialogue qui apparaîtra après que l'on ait branché le mobile à l'ordinateur par USB.

Sous Linux et Mac, aucun driver n'est à installer.

Le répertoire « platforms » contient entre autres la bibliothèque Java android.jar. Ce jar contient toutes les classes Java (Java et non au format dex) constituant le SDK. En examinant le contenu du jar, on s'apercevra qu'il contient aussi bien les classes des packages propres à Android (android.*) que les classes du JDK (java.*). Il ne faut pas en effet oublier qu'une compilation Android remplace le bootclasspath.

Le dossier « add-ons » accueille les API optionnelles de la plateforme comme par exemple Google Maps.

Enfin, à la racine du SDK, se trouve le fichier documentation.HTML ouvrant sur la documentation, la note de version (RELEASE_NOTES.html).

3.2.1 Plug-in Eclipse

Les outils

Les outils Android sont localisés dans le répertoire « tools » du SDK. Si le développement se fait avec le *plug-in* Eclipse, la majorité de ces utilitaires s'utiliseront de façon transparente sans même que le programmeur ne s'en rende compte. La génération des objets ressources, la transformation des classes java en classes Dalvik (format dex), la création du package apk puis son déploiement sur l'émulateur se font par exemple simplement en lançant l'application depuis Eclipse.

La connaissance précise du fonctionnement de ces outils n'est donc pas indispensable pour commencer à construire son application. Toutefois, lorsque des besoins plus poussés tels que la simulation de réception de SMS ou de carte SD apparaîtront, il faudra se pencher sur la documentation de ces outils !

L'émulateur Android

L'émulateur se lance par la commande « emulator ». Celle-ci prend en paramètre l'image AVD (*Android Virtual Device*) qui sera montée en mémoire. Il est donc possible de préparer et de conserver de multiples images qui sont autant de téléphones virtuels, ce qui est très commode pour les tests. La commande accepte un nombre important d'options qui permettent de spécifier des images ramdisk ou de simuler la présence de sdcard.

Il est possible de lancer plusieurs émulateurs en même temps, ce qui est utile pour tester des applications faisant interagir deux combinés entre eux. Toutefois, aucune communication ne peut se faire directement d'émulateur à émulateur sans une configuration préalable car ils sont isolés derrière un routeur pare-feu virtuel.

Chaque émulateur écoute un port dont le numéro apparaît sur la barre de la fenêtre. Par Telnet on peut se connecter ainsi à la console de l'émulateur et exécuter certaines opérations de configuration.

Par exemple : telnet localhost 5554

Il n'est pas possible de décrire ici toutes les possibilités qu'offre l'émulateur tant elles sont nombreuses. En outre, généralement, on n'a besoin que ponctuellement de se servir de ces options. Dans ces cas précis heureusement, la documentation de Google est exhaustive : <http://developer.android.com/guide/developing/tools/emulator.html>

activitycreator

activitycreator est un script, un batch sous Windows et un shell sous Linux, qui lance une application *standalone* Java fonctionnant uniquement en ligne de commande sans interface graphique.

Cette commande a pour but de créer un squelette de projet Android. Si l'on utilise Eclipse, cette commande n'a pas vraiment d'intérêt car une structure de projet, qui plus est, spécifiquement adaptée à l'IDE, est déjà créée par le *plug-in*.

Malgré tout, même avec Eclipse, activitycreator pourrait être finalement utile car s'il est vrai qu'activitycreator ne crée pas les fichiers « .classpath » et « .project »

propres à Eclipse, *activitycreator* crée par contre le fichier *ant build.xml* ce que ne fait pas le *plug-in*.

Pour son usage, *activitycreator* attend un unique argument qui est le nom qualifié de la classe de l'activité principale. Par l'option « *-out* », il est possible de fournir le répertoire de génération des répertoires et fichiers projet (par défaut il s'agit du répertoire courant où est exécutée la commande). Enfin l'option « *-ide intellij* » indique à l'outil que le projet est destiné à être développé avec l'environnement *intellij* (concurrent d'Eclipse à l'excellente réputation) et qu'il convient donc de générer le fichier projet à son format.

aapt

La commande « *aapt* », qui signifie *Android Asset Packaging Tool*, sert à manipuler les packages des applications Android, fichiers ayant pour extension *apk*. Les fichiers *apk* comme les fichiers standard *jar* (Java ARchive) sont des fichiers compressés qui regroupent l'ensemble des éléments constituant le programme.

Le rôle majeur d'« *aapt* » est de compiler les ressources des applications Android. Cette opération consiste à transformer certains artefacts qui ne sont pas des fichiers sources Java, par exemple des fichiers *xml* décrivant des animations, dans leur équivalent binaire (fichiers « *class* » de bytecode).

Comme la plupart des commandes Android, l'exécution de cette commande se fait automatiquement par le *plug-in* Eclipse.

aidl

« *aidl* » (*Android Interface Definition Language*) fonctionne dans le même esprit que la commande *idlj* du JDK standard.

Plus précisément, « *aidl* » prend en entrée un fichier de description d'interface (dans le format *aidl* donc, qui est propriétaire à Android et qui ressemble quelque peu à l'IDL de Corba) et génère en sortie les fichiers sources Java qui serviront à la fois au client de l'interface décrite et à la construction de l'implémentation de cette même interface.

adb

Grâce à la commande « *adb* » (*Android Debug Bridge*), on peut communiquer avec le terminal pour lui envoyer des instructions. « *adb* » fonctionne selon un modèle client/serveur. Les émulateurs, comme on le sait maintenant, ouvrent un port de connexion Telnet (par défaut le 5554) ; en réalité ils en ouvrent aussi un deuxième (par défaut le 5555) pour la communication avec *adb*. Il est à remarquer qu'*adb* fonctionne aussi avec les vrais téléphones.

Pour ce qui est du détail des commandes, là aussi, le choix est vaste :

- « *adb install* », suivi du fichier *apk*, déploie l'application packagée.
- « *adb pull* » et « *adb push* » copie un fichier depuis et vers l'émulateur (ou le téléphone).

- « adb shell » initie une connexion shell sur le terminal grâce à laquelle on pourra exécuter directement sur le système Android les commandes localisées dans le répertoire /system/bin du téléphone. Ce répertoire contient, parmi d'autres, le client `sqlite3` qui sert à consulter les bases SQL locales ou `monkey` qui génère des événements utilisateurs comme les clics ou les mouvements tactiles sur l'écran.

Enfin, avec `adb`, on peut visualiser les logs et contrôler leur production.

dx

Le but principal de `dx` est de prendre en entrée des classes au format bytecode JVM et d'en sortir des fichiers au format binaire dex (Dalvik Executable). L'exécution de la commande suivie de « `-help` » renseigne sur les options relativement nombreuses de l'outil. Certaines options servent à contrôler la manière dont les fichiers dex sont générés (avec ou sans optimisation, inclusion ou non des noms de variables locales...) et aussi à exécuter des tests `junit`.

La documentation fournie dans le sdk de la commande est très succincte et il faut souvent deviner (en faisant des analogies avec les options de `javac`) le sens de tel ou tel paramètre de commande. Que le développeur néanmoins se rassure, ces options n'ont qu'une importance très limitée.

apkbuilder

`apkbuilder` est la commande (elle-même écrite en java) à utiliser pour générer les package apk Android. C'est la dernière étape nécessaire avant de pouvoir déployer son application sur le terminal. En paramètre de cette commande, il faut indiquer les chemins vers les fichiers (ressources, dex...) qui seront assemblés pour créer le package. Cette commande est aussi capable de signer le package avec une clé de test (*debug key*), ce qui est exigé pour pouvoir installer l'application sur l'émulateur. Par contre, pour distribuer l'application finale, il faudra la signer avec une clé privée classique et cela ne se fera pas avec `apkbuilder` mais avec la commande `jarsigner` du JDK.

sqlite3

Il s'agit de la commande `sqlite` normale dont la documentation officielle se trouve à l'adresse : <http://www.sqlite.org/sqlite.html>

L'objet de `sqlite3` est d'exécuter des commandes d'administration `sqlite3` et même des requêtes SQL sur une base de données spécifiées en argument. Il s'agit bien sûr d'une base `sqlite` locale au téléphone et non d'une base distante hébergée sur un serveur. Le nom de la base se précise par le nom du fichier de stockage.

Pour utiliser `sqlite3`, il faut préalablement avoir ouvert un « remote shell » sur un émulateur ou sur un véritablement téléphone (`adb shell`).

3.2.2 Scripts Ant

Comme expliqué ci-dessus, la commande `activitycreator` génère un embryon de projet utile pour un démarrage rapide. Parmi les artefacts produits, le fichier « `build.xml` » mérite d'être conservé à double titre.

Tout d'abord, être en capacité de générer le package complet de l'application en dehors d'Eclipse est indispensable pour la mise en place d'un processus d'intégration continue.

Ensuite, du point de vue pédagogique, la lecture du script ant est très instructive. En effet, en examinant les dépendances entre les targets ant, on comprend bien la succession des commandes à enchaîner pour construire et déployer un programme Android.

3.3 DÉVELOPPER, EXÉCUTER & DÉBUGGER

3.3.1 Installation

Pour commencer à développer, il faut s'équiper :

- Le **JDK**, on peut le trouver là <http://java.sun.com/javase/downloads/index.jsp>, la documentation Javadoc mérite aussi d'être téléchargée.
- **Eclipse**, le simple bundle « Eclipse IDE for Java Developers » suffit, <http://www.eclipse.org/downloads/>
- Le **SDK Android**, <http://developer.android.com/>
- Le **plug-in Eclipse** d'Android (ADT)

Le SDK Android est un zip, qu'il convient de décompresser. On peut alors ajouter à son path le répertoire tools contenant les utilitaires en ligne de commande. Sur un système Unix/Linux ça se passe dans le `~/.bash_profile` (pour le shell bash ou ksh ce serait `.profile`) ; sous Windows c'est dans le « Panneau de configuration/Système ». Dans les paramètres avancés, un clic sur « Variables d'environnement... » ouvre la fenêtre de la figure 3.2.

La variable Path se configure de façon globale (« Variables système »), ou seulement pour l'utilisateur connecté ; sous un OS Unix/Linux, le fichier `/etc/profile` est l'emplacement pour les variables applicables à tous. Au passage, il est aussi de bon ton d'ajouter au path le répertoire bin du JDK.

Pour le *plug-in* Eclipse, il faut directement le télécharger depuis l'IDE. Pour cela, il faut, après avoir démarré Eclipse, cliquer sur le menu « Help/Software Updates... », puis ajouter à la liste des sites le dépôt <http://dl-ssl.google.com/android/eclipse/>. On peut alors installer les deux modules "Android Developer Tools" et "Android Editors".

Enfin pour parachever l'installation, dans Eclipse, menu « Windows/Preferences », il faut indiquer que le répertoire où le SDK Android a été dézippé.



Figure 3.2 — Les variables d'environnement sous Windows

3.3.2 Dalvik Debug Monitor Service (DDMS)

Le SDK Android vient avec un outil de débogage nommé DDMS (*Dalvik Debug Monitor Service*). Cet outil peut se lancer par le batch `ddms.bat`, il fonctionnera alors de façon autonome. Il peut aussi s'utiliser au travers d'Eclipse et de la perspective DDMS.

DDMS est très complet et présente à l'utilisateur de nombreuses informations d'inspection sur la machine virtuelle Dalvik :

- la liste des threads et leur état,
- l'activité du Garbage Collector,
- l'ensemble des logs du système...

En outre, DDMS, qui s'appuie également sur `adb`, est capable de piloter le terminal ou le téléphone pour simuler des appels téléphoniques entrants, la réception de SMS ou déterminer la position GPS du terminal. DDMS est en fait le compagnon graphique des outils Android, indispensable en phase de développement.

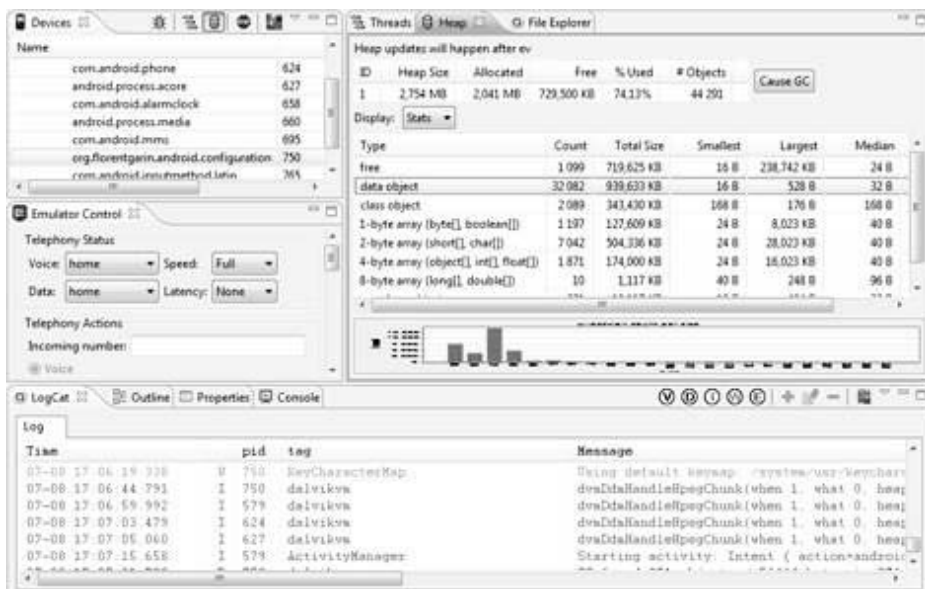


Figure 3.3 — La perspective DDMS

3.4 STRUCTURE D'UN PROJET

La structure d'un projet Android répond à des règles bien précises. Le meilleur moyen pour les appréhender est de commencer par créer un projet de test à l'aide du *plug-in* Eclipse d'Android.

Le menu Eclipse « File/New/Other... » amène à la boîte de dialogue depuis laquelle l'assistant de création de projet Android est accessible (figure 3.3).

- Les champs « Project name » et « Package name » sont sans surprise : il s'agit simplement du nom du projet Eclipse et du package Java racine de l'application.
- Le champ « Activity name » est typique à Android.

Le concept d'Activité est très important dans Android et il convient de parfaitement l'appréhender. Cette notion est la base de l'extraordinaire capacité de coopération des applications.

Les Activités ont un chapitre entier qui leur est consacré plus loin. Pour le moment, de manière simpliste, une Activité pourrait être définie comme le point d'entrée, équivalent de la classe contenant la méthode static main, d'une application Java de base. L'activité est aussi le point d'ancrage, où est défini le contenu visuel de l'écran. En ce sens et pour continuer les analogies, la classe *Activity* se rapproche de la classe java.applet.Applet qui elle est le point de démarrage des applications Java exécutées au sein même des pages HTML.

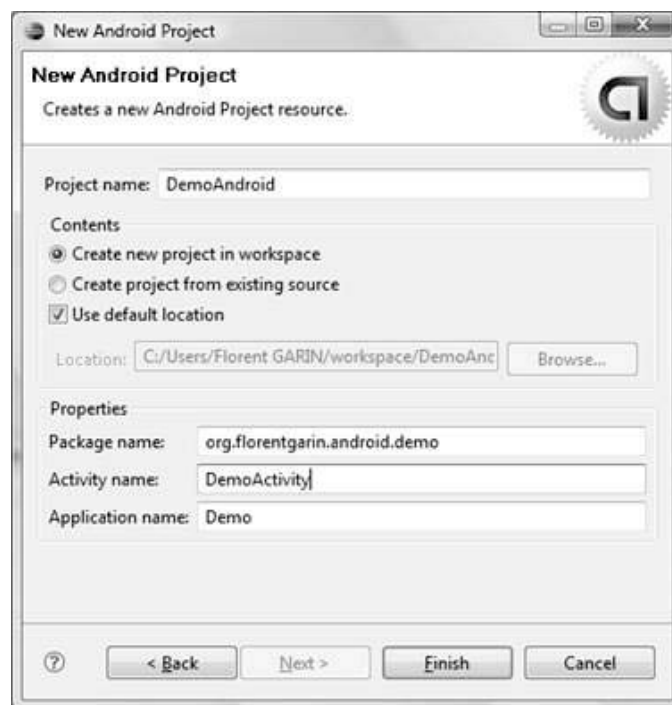


Figure 3.4 — Assistant Eclipse de création de projets

Une fois créé, le projet possède la structure arborescente suivante :

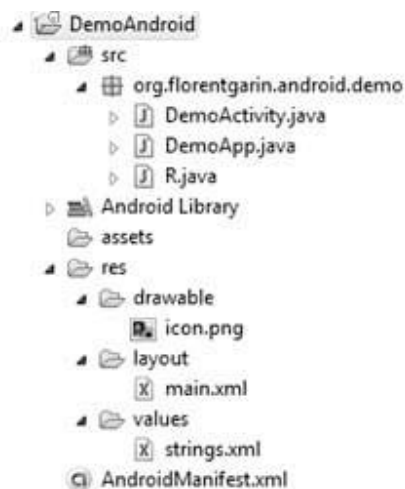


Figure 3.5 — Arborescence initiale

3.4.1 Le manifest

Le fichier `AndroidManifest.xml` est, comme son nom l'indique, le fichier manifeste d'Android au format xml !

Fichier manifest

Un manifeste est un petit fichier qui contient des informations sur l'application ou la librairie à laquelle il appartient. Regrouper ainsi des métadonnées dans un fichier, qui est au format xml dans le cas d'Android mais ce n'est pas une obligation, est une pratique courante dans le monde Java.

Plus sérieusement, ce fichier regroupe les éléments de configuration de l'application. Il est à Android ce que le `web.xml` est aux applications web Java. Il peut se modifier à l'aide de l'assistant du *plug-in* Eclipse d'Android ou en modifiant directement le fichier source xml. En double-cliquant sur le fichier depuis la vue « Package Explorer », la fenêtre suivante apparaît :

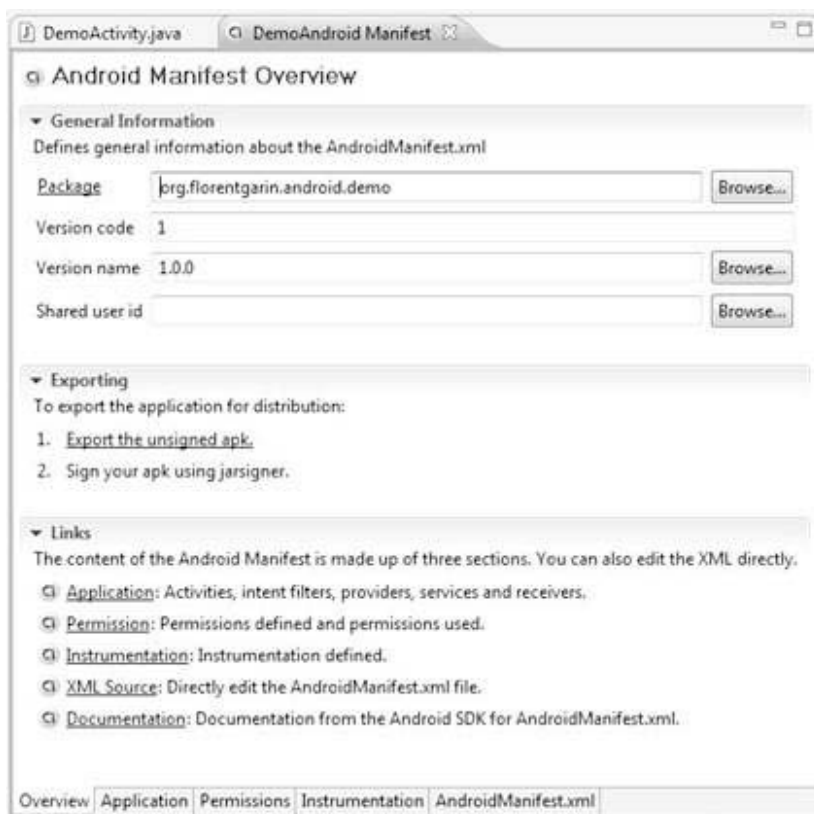


Figure 3.6 — Onglet "Overview"

Cette fenêtre sert à éditer le fichier manifeste de façon graphique. Les données sont subdivisées en quatre catégories :

- Overview
- Application
- Permissions
- Instrumentation (ce dernier onglet présente la vue xml du fichier).

Certaines propriétés peuvent être renseignées directement dans le champ de texte correspondant de la fenêtre de l'éditeur de manifeste. Par exemple, sur l'onglet « Application », le champ label peut être rempli librement. D'autres, au contraire, doivent forcément être valorisées au travers d'un fichier de ressources. Toujours dans le même onglet, la propriété description, par exemple, doit obligatoirement faire référence à une entrée dans un fichier de ressources (pseudo-fichier *properties*). Enfin certaines pointent vers un fichier de ressource comme l'attribut « icon » de l'application qui fait référence à une image placée dans le répertoire res/drawable.

Pour s'assurer de correctement éditer le fichier AndroidManifest.xml, il est donc préférable de se servir de l'assistant.

Overview

Les champs à saisir concernent les paramètres globaux de l'application : dans le fichier xml, cela représente le tag racine « manifest ».

Tableau 3.1 — Les champs de l'onglet « Overview »

Nom du champ	Description
Package	Le nom du package racine de l'application. Il identifie unitairement l'application. Cette valeur suit les règles de nommage des packages Java : seuls les caractères de type lettre ASCII en minuscules sont valables. Pour s'affranchir des problèmes de collisions de noms, la pratique veut que le début du package commence par le nom de domaine inversé de l'organisation responsable de l'application.
Version code	Le « Version code » est un entier représentant la version de l'application. Ce champ est important car c'est celui qui permet de comparer deux versions d'un même logiciel et de déterminer ainsi si une mise à jour est nécessaire. L'évolution de ce champ est par contre libre, on peut imaginer qu'il soit incrémenté à chaque livraison de l'application mais il n'y a aucune obligation, sa progression n'est pas tenue d'être linéaire.
Version name	Ce champ est libre, et peut prendre n'importe quelle valeur de type String. Il s'agit de la version du logiciel affichée à l'utilisateur. Cela peut correspondre à la version commerciale ; contrairement au code de version, ce champ n'a pas d'incidence technique.
Shared user id	Ce champ facultatif est l'équivalent du compte utilisateur Linux avec lequel sera exécutée l'application. S'il est omis, Android attribue automatique un « user id » unique à chaque application afin d'obtenir un bon niveau de sécurité et de les isoler les unes des autres. Par contre, si deux applications ont le même « Shared user id », elles pourront partager les données. Pour cela, il faut, en plus de définir la même valeur pour cette entrée, qu'elles aient été signées avec le même certificat. Posséder un « Shared user id » identique est la condition préalable pour configurer deux applications à tourner dans le même processus Linux VM Dalvik.

Application

L'onglet « Application » gère graphiquement l'édition du tag « application » et de ses sous-éléments correspondant au paramétrage des « Application Nodes ». Dans le fichier xml, le tag « application » est lui-même un sous-élément de « manifest ».



Figure 3.7 — Onglet "Application"

Tableau 3.2 — Tableau 3.2 Les champs de l'onglet « Application »

Nom du champ	Description
Application Toggle	Cliquer sur cette case à cocher, supprime le tag « application » du fichier xml ainsi que tous ses sous éléments. Cela a pour effet d'annihiler toutes les options de paramétrage renseignées dans cet onglet.
Name	Ce champ optionnel renseigne le nom de la sous-classe d'android.app.Application que l'on souhaite utiliser. La classe Application, ou donc une sous-classe que l'on aurait soi-même créée, est automatiquement instanciée au démarrage. Cette classe permet de stocker les variables globales du programme à la façon de l'« application context » des webapp java.
Theme	Définit le thème visuel (skin) de l'application. Le thème peut être redéfini précisément pour chacune des activités constituant l'application. Le thème est en quelque sorte l'équivalent du PLAF (<i>Pluggable Look and Feel</i>) de Java Swing.
Label	Il s'agit juste du nom de l'application. Ce nom sera celui affiché à l'utilisateur.
Icon	L'icône de l'application. Le fichier ressource image référencé doit bien sûr exister.
Description	Ce champ, comme le label, est purement informatif : il décrit en détail l'application.
Permission	Ici sont listées les permissions qui devront être accordées à l'application pour fonctionner correctement. Par exemple si une application nécessite d'effectuer des requêtes sur Internet, il faudra déclarer la permission qui a pour nom : « android.permission.INTERNET ».
Process	<p>Le nom du processus Linux sous lequel tourneront les composants (les objets <i>Activity</i>, <i>Service</i>, <i>BroadcastReceiver</i>, <i>ContentProvider</i>) de l'application. Par défaut, il s'agit de la valeur du champ « Package » de l'onglet « Overview ». Si cette valeur n'est pas redéfinie au niveau du composant, ces derniers seront tous exécutés dans le même processus. Dans le cas contraire, ils pourront être isolés en étant portés par des processus différents.</p> <p>Il est aussi possible de mutualiser un même processus entre plusieurs composants issus de plusieurs applications. Pour cela, la valeur de l'attribut « Process » doit être bien entendu la même mais ils doivent aussi partager le même utilisateur Linux, c'est-à-dire avoir le même « Shared user id ». Si le nom du processus commence par « : », ce processus sera privé à l'application et ne pourra en aucun cas être partagé avec d'autres composants d'autres applications.</p>
Task affinity	<p>La notion de Task dans Android doit être appréhendée sous l'angle de vue purement utilisateur et non selon une perspective technique. Une « Task » représente une pile d'activités, c'est-à-dire d'écrans, ordonnée à la manière d'un historique de navigation web.</p> <p>L'attribut affinité, indique dans quelle pile les écrans (les activités) sont destinés à être rangés. Si l'attribut est défini au niveau de l'application alors toutes les activités de celle-ci hériteront de cette valeur, ce qui signifie qu'elles seront conçues pour être classées dans la même pile d'historique. Cet attribut peut aussi être individuellement défini ou redéfini sur l'activité ce qui signifie que les activités d'une application ne seront pas forcément classées dans la même task.</p>

Tableau 3.2 — (suite)

Allow task reparenting	Valeur « true » ou « false ». Cet attribut est à rapprocher de « task affinity », il concerne également la notion de task. Il s'agit ici de renseigner si l'activité pourra être déplacée de la « task » de l'activité précédente ayant lancé la nouvelle activité vers la « task » correspondant à son affinity.
Has code	Valeur booléenne qui détermine si le système doit essayer de charger du code en mémoire au moment du lancement des composants de l'application. En d'autres termes, si « Has code » égale « false », cela veut dire que l'application ne contient aucun code binaire à exécuter ! Cela peut sembler étrange de déployer une application vide : qu'on se rassure, la valeur par défaut de « Has code » est « true », mais il existe un mécanisme par lequel on peut déclarer dans une application, une activité comme étant un alias d'une autre activité provenant d'une autre application.
Persistent	Booléen indiquant si l'application doit tourner en permanence. La valeur par défaut est « false ». Il est préférable, en effet, de laisser le système démarrer et stopper les processus Linux. La VM Dalvik est prévue pour pouvoir se lancer très rapidement, de plus une application permanente consomme inévitablement des ressources système. En conclusion, à moins d'avoir une raison bien particulière, il vaut mieux ne pas positionner ce champ à vrai.
Enabled	Active (« true » valeur par défaut) ou désactive (« false ») les composants de l'application. Chaque composant pourra, au niveau de ses propres propriétés, définir l'attribut « enabled ». Un composant ne sera considéré comme actif que si sa propre propriété « enabled » vaut true ainsi que celle plus globale de l'application. La notion de composant est décrite dans un autre chapitre.
Debuggable	Les valeurs possibles sont « true » ou « false » (par défaut). Ce champ autorise ou non de déboguer l'application (positionner des points d'arrêt, faire du pas à pas dans le code, inspecter les valeurs des variables...). Le débogage est possible aussi bien depuis l'émulateur que depuis un vrai téléphone.
Manage space activity	Le nom qualifié (avec le nom de package) de la classe de l'activité à afficher lorsque l'utilisateur voudra libérer la mémoire occupée par l'application. C'est par le programme « Application Manager », inclus de base dans Android, que cette activité sera lancée.
Allow clear user data	Prend la valeur « true » (valeur par défaut) ou « false ». Indique si oui ou non, l'utilisateur peut effacer les données de l'application à l'aide de l'« Application Manager ». Le gestionnaire d'applications est un programme accessible directement depuis le téléphone grâce auquel on peut examiner les propriétés des applications installées, comme les permissions qu'elles requièrent ou l'espace de stockage utilisé, ou encore effacer leurs données quand cela est autorisé.
Application Nodes	Les « applications nodes » sont les composants centraux des applications. Il en existe de quatre types : <i>Activity</i> , <i>Service</i> , <i>BroadcastReceiver</i> et <i>ContentProvider</i> . Il convient ici de lister et de paramétrer ceux faisant partie de l'application. Ces composants sont étudiés en profondeur dans d'autres chapitres.

Permissions



Figure 3.8 — Onglet « Permissions »

Sur l'onglet *Application*, il était possible de définir une et une seule permission requise par le programme. Ici, sur cet onglet consacré uniquement à ce sujet, on peut définir plusieurs permissions requises par l'application grâce à l'élément `uses-permission`.

Mais on peut aussi faire l'inverse, c'est-à-dire déclarer les permissions qui devront avoir été accordées aux autres applications qui voudront interagir avec l'application du manifeste (élément `permission`).

Instrumentation

Cet onglet permet d'installer et de configurer des éléments d'instrumentation dans l'application (tag xml « instrumentation »). C'est grâce aux instrumentations qu'on peut dérouler des tests unitaires réalisés avec le framework JUnit (figure 3.9 et tableau 3.3).

Les métadonnées

Des valeurs de paramètres de quelque nature qu'elles soient peuvent être transmises aux composants via la balise `<meta-data>`. Elles sont ensuite récupérées dans le code un peu comme les paramètres applicatifs des servlets configurés dans le `web.xml`.

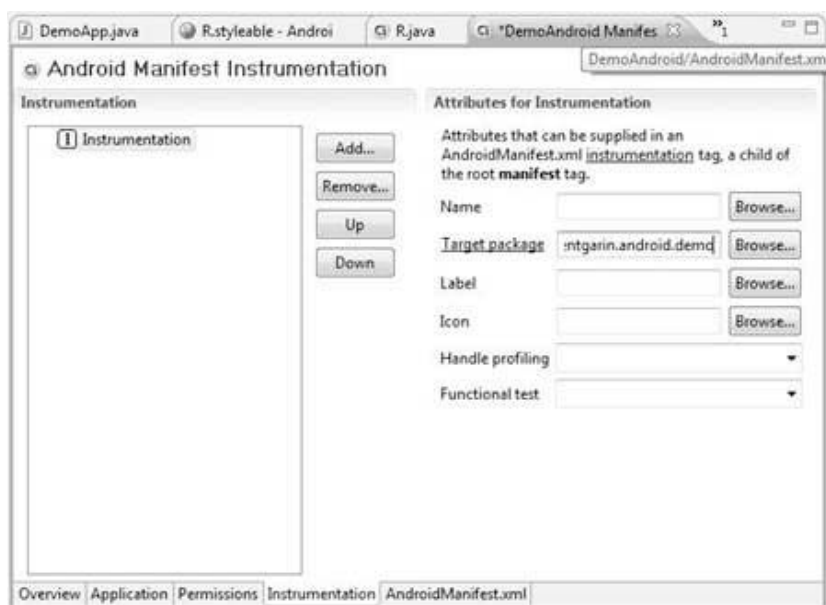


Figure 3.9 — Onglet "Instrumentation"

Tableau 3.3 — Les champs de l'onglet « Instrumentation »

Nom du champ	Description
Name	Le nom (« fully qualified ») de la classe d'instrumentation que l'on souhaite installer dans l'application. Cette classe doit nécessairement étendre la classe <code>android.app.Instrumentation</code> . <code>InstrumentationTestRunner</code> est une spécialisation de la classe de base <code>Instrumentation</code> dédiée aux compagnes de tests. Plusieurs classes d'instrumentation peuvent être ajoutées à l'application. Si ce nom commence par un point « . », alors le nom de l'application, attribut <code>Package</code> du tag « <code>manifest</code> », servira de préfixe pour obtenir le nom qualifié de la classe à instancier.
Target package	Ce champ contient l'id de l'application sur laquelle appliquer l'instrumentation courant. Pour rappel, l'id d'une application est son package racine.
Label	Un simple label permettant d'identifier visuellement l'instrumentation.
Icon	Une image référençant l'instrumentation. L'icône, comme le label, ne sert que de repère visuel.
Handle profiling	Indique si la classe d'instrumentation aura la main sur l'activation ou non du profiling.
Functional test	Ce booléen, s'il prend la valeur vrai, notifie que le test représenté par l'instrumentation est un test fonctionnel. Il aura donc besoin d'un contexte Android complet et conforme à celui obtenu lorsque l'application est exécutée normalement. Via les composants d'instrumentation, il est alors possible de générer des événements utilisateurs et de contrôler les réponses des éléments d'IHM.

3.4.2 Les ressources

Les ressources sont des éléments constituant l'application, hormis le code source lui-même. Il peut s'agir de fichiers image, vidéo, du texte...

Il y a deux grandes familles de ressources, chacune ayant son propre répertoire de stockage au sein des applications Android : « res » et « assets ».

« res » abritera les ressources typées et référencées par Android alors que le dossier « assets » accueillera les fichiers bruts qui seront lus sous forme de flux d'octets indifféremment de la nature même de leur contenu.

Les ressources « res »

Les ressources « res » sont disposées dans des sous-répertoires de « res » en fonction de leur type. Ces ressources sont référencées dans la classe spéciale `R`, étudiée ci-après. Cela veut dire que cette fameuse classe définira des constantes de type `int` permettant l'identification de chacune de ces ressources. Ces ID sont utiles à de nombreux endroits dans l'API Android.

Ces ressources sont également typées : en effet leur ventilation dans les sous-répertoires se fait très précisément. La raison de cette façon de faire est que lors du processus de packaging de l'application, les ressources sont transformées en binaire pour des questions d'optimisation. Un fichier `jpg` sera par exemple compilé en un objet `Drawable`. Il convient donc de bien catégoriser les ressources.

Par défaut, l'assistant de création de projet Android d'Eclipse crée trois dossiers « res » prêts à accueillir les ressources les plus fréquemment employées :

- **res/drawable** - Ce dossier stocke les images `jpg`, `png` et même `gif` (format néanmoins obsolète qu'il est préférable d'éviter). Ces images deviendront donc des `Drawable` et pourront être affichées à l'écran.
- **res/layout** - Ce répertoire contient les fichiers `xml` décrivant l'agencement de l'IHM : la disposition des widgets, leur taille, style à appliquer... Le format des fichiers `layout` est vu au chapitre « Les composants graphiques ».
- **res/values** - Ce répertoire héberge toutes les ressources exprimées textuellement sous forme de fichier `xml`. Ces ressources peuvent toutes être déclarées dans un seul et même fichier ou réparties selon un découpage libre dans plusieurs fichiers `xml`. La convention cependant dicte plutôt de rassembler dans un même fichier les ressources de même catégorie, ce fichier portant le nom de la catégorie. Par exemple, les simples chaînes de caractères devraient toutes être dans le fichier « `strings.xml` », les couleurs dans « `colors.xml` », les tableaux dans « `arrays.xml` »...

En plus de ces trois répertoires créés par le *plug-in* Eclipse, il est possible d'en rajouter d'autres qui contiendront d'autres types de ressource :

- **res/anim** - Android offre un langage d'animation minimaliste au format `xml`. Ce répertoire est le lieu où il convient de les stocker. Le format de ces fichiers est expliqué dans un paragraphe qui lui est consacré dans la partie « Les animations ».

- **res/menu** - Les menus de l'application, qu'il s'agisse des menus d'options (affichés lorsqu'on presse le bouton MENU) ou contextuels (en l'absence de clic droit, ils apparaissent suite à un clic tenu de plusieurs secondes), peuvent se déclarer par des fichiers xml mis dans le répertoire « res/menu ». La grammaire de définition des menus est abordée plus en détail au paragraphe qui lui est consacré.
- **res/xml** - C'est le répertoire pour placer les fichiers xml applicatifs dont le format n'est pas connu d'Android. Ces fichiers seront lus dans le code de l'application au travers de l'objet *XmlResourceParser* obtenu par la méthode *Resources.getXml(int id)*. L'objet *Resource* sera lui-même récupéré à partir de l'objet *Context*. *XmlResourceParser* est un parseur de type XMLPull : cette nouvelle famille de parseurs tente de faire la synthèse de ce qu'il y a de meilleur dans les parseurs SAX et DOM. En effet, ces nouveaux parseurs ont une faible empreinte mémoire et sont capables de parcourir des fichiers très volumineux tout en exposant une API élégante et pratique pour le développeur.
- **res/raw** - Les fichiers contenus dans le sous-répertoire « raw » ne seront pas compilés mais se retrouveront tels quels dans le package apk. Ces fichiers seront lus par l'interface *InputStream* obtenue par *Resources.openRawResource(int id)*. En somme, ces ressources ont un mode opératoire assez similaire aux « assets » à la différence notable que les fichiers « res/raw » ont un identifiant répertorié dans la classe R.

Les ressources « assets »

Les éléments de type « assets » (signifie en anglais « actif », comme une machine outil fait partie des « actifs » d'une entreprise) ne sont pas traités par les outils de package d'Android : ils seront copiés sans transformation sur le terminal au moment du déploiement et n'auront pas de constante int dans la classe R les identifiant.

La lecture du contenu des fichiers se fait grâce à l'objet *AssetManager* qui, en lui soumettant le nom du fichier par la méthode *open*, renvoie un *InputStream*. Pour obtenir une référence vers l'objet *AssetManager* de l'application, il faut appeler *Context.getAssets()*.

La classe R

La classe R, 'R' comme « Resources », est une classe générée au moment de la création du fichier .apk, qui est le fichier de l'application packagée, par l'utilitaire aapt (Android Asset Packaging Tool) qui se trouve comme les autres utilitaires dans le répertoire « tools » du SDK Android.

Ceux qui feront le choix d'Eclipse peuvent se rassurer : le *plug-in* régénère automatiquement la classe R dès que des changements sur le projet le nécessitent, c'est-à-dire dès que des modifications, suppressions ou ajouts sont apportés aux fichiers présents dans les sous-répertoires de « res ». Les ressources positionnées dans le dossier « assets » ne sont pas concernées car elles ne sont pas référencées par Android.

Voici le genre de contenu que la classe R pourra avoir :

```
/* AUTO-GENERATED FILE. DO NOT MODIFY.
 *
 * This class was automatically generated by the
 * aapt tool from the resource data it found. It
 * should not be modified by hand.
 */
package org.florentgarin.android.widgets;
public final class R {
    public static final class array {
        public static final int saisons=0x7f040000;
    }
    public static final class attr {
    }
    public static final class drawable {
        public static final int gallery1=0x7f020000;
        public static final int gallery10=0x7f020001;
        public static final int gallery2=0x7f020002;
        public static final int gallery3=0x7f020003;
        public static final int gallery4=0x7f020004;
        public static final int gallery5=0x7f020005;
        public static final int gallery6=0x7f020006;
        public static final int gallery7=0x7f020007;
        public static final int gallery8=0x7f020008;
        public static final int gallery9=0x7f020009;
        public static final int icon=0x7f02000a;
        public static final int logo=0x7f02000b;
    }
    public static final class id {
        public static final int CancelButton=0x7f060002;
        public static final int bold_cb=0x7f060005;
        public static final int clock=0x7f060007;
        public static final int color=0x7f06000e;
        public static final int gallery=0x7f06000b;
        public static final int image=0x7f06000c;
        public static final int italic_cb=0x7f060006;
        public static final int okButton=0x7f060001;
        public static final int option1=0x7f06000f;
        public static final int option2=0x7f060010;
        public static final int option3=0x7f060011;
        public static final int option4=0x7f060012;
        public static final int password=0x7f06000a;
        public static final int phoneNumber=0x7f060009;
        public static final int plain_cb=0x7f060003;
        public static final int progress=0x7f06000d;
        public static final int saisons=0x7f060013;
        public static final int serif_cb=0x7f060004;
        public static final int time=0x7f060014;
        public static final int title=0x7f060008;
        public static final int toggle1=0x7f060015;
        public static final int toggle2=0x7f060016;
        public static final int voiture=0x7f060000;
    }
    public static final class layout {
        public static final int analog_clock=0x7f030000;
        public static final int auto_complete_text_view=0x7f030001;
```

```

        public static final int buttons=0x7f030002;
        public static final int check_boxes=0x7f030003;
        public static final int date_picker=0x7f030004;
        public static final int digital_clock=0x7f030005;
        public static final int edit_texts=0x7f030006;
        public static final int gallery=0x7f030007;
    }
    public static final class string {
        public static final int app_name=0x7f050001;
        public static final int hello=0x7f050000;
        public static final int prompt=0x7f050002;
    }
}

```

La classe R est frappée du modificateur « final », elle ne peut donc pas être étendue. Rien d'étonnant à cela, sa raison d'être étant simplement de regrouper les identifiants des ressources du projet, cette classe ne définit aucune méthode ni aucun attribut. Par contre, la classe R déclare des classes internes statiques et finales comme « drawable », « id », « layout »...

Ce sont ces classes qui encapsulent véritablement les constantes de type int. Les règles de codage Java stipulent que le nom des classes doit commencer par une lettre majuscule ce qui n'est pas le cas ici parce que le pattern de programmation utilisé ici cherche uniquement à regrouper les constantes selon leur catégorie.

Utilisation des ressources

C'est bien beau de créer des composants ressources et de les copier sous « res » mais il faut maintenant les utiliser. Comment faire ?

Dans le code Java, c'est très simple : l'API Android accepte partout où les ressources pourraient être employées le moyen d'y faire référence en spécifiant un paramètre de type int qui ne saurait être autre chose que les id répertoriés dans la classe R.

Par exemple, pour positionner une image de fond sur un bouton, il suffit d'appeler la méthode `setBackgroundResource(int id)` sur le widget en passant en paramètre la constante `R.drawable.XXX` qu'il convient.

À l'usage, cette facilité d'emploi est appréciable ainsi que la robustesse du code qui en découle. En effet, il n'y a pas de risque de faire une faute de frappe sur le nom de la ressource ou de pointer vers un élément qui n'existe plus. C'est une évolution favorable du « `ClassLoader.getResourceAsStream()` » de Java qui évite le syndrome des liens cassés.

Si toutefois on souhaite récupérer l'objet représentant la ressource et non pas l'affecter directement ; pour y appliquer quelques traitements préalables par exemple, il faudra se servir de la classe *Resources*, par exemple :

```

Resources res = getResources();
Drawable logo = res.getDrawable(R.drawable.logo);

```

Resources possède de multiples méthodes qui renvoient les objets correspondant aux types de ressources (`getXml(int id)`, `getText(int id)`...).

Parfois, il peut arriver qu'on veuille utiliser une ressource dans la définition même d'une autre. Pour cela, il y a une notation spéciale de la forme :

@[nom du package :]type de la ressource/nom de la ressource

Par exemple, dans le fichier manifeste au niveau du tag « application », l'attribut « android:label », qui indique le nom de l'application, a la valeur suivante (avec une création de projet par l'assistant) « @string/app_name ». Le nom du package a été omis car la ressource appartient à la même application. Cette ressource est présente dans le fichier « res/values/strings.xml » dont le contenu est :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Hello World, DemoActivity</string>
    <string name="app_name">Demo</string>
    <string name="app_description">Application de test</string>
</resources>
```

Ce fichier « strings.xml » suit la convention de nommage qui veut que l'on déclare au même endroit les ressources de même type dans des fichiers portant le nom du type (avec un « s »). Cependant, il faut bien noter que la référence « @string/app_name » ne dépend pas du nom de stockage du fichier mais bien du type.

Il existe une autre notation, avec un « ? » à la place du « @ ». Elle est utilisée pour pointer vers un attribut de thème.

Pour définir le style graphique d'une barre de progression on peut, par exemple, ajouter dans le fichier layout sur le tag ProgressBar l'attribut xml :

```
■ style="?android:attr/progressBarStyleHorizontal"
```

Un paragraphe plus bas, dans la section « Fonctionnalités avancées », est consacré aux thèmes.

3.5 ET LES WEBAPP ?

Une application Android est, pour reprendre un terme communément employé pour les applications de bureau, une application lourde. En effet, le code compilé, les diverses ressources de l'application et autre fichier manifeste sont packagés en un seul fichier apk qui est déployé ensuite sur le terminal.

Un programme Android n'est donc pas une webapp¹. Les web applications ont un avantage de taille par rapport aux applications classiques : aucun déploiement préalable sur le poste de l'utilisateur n'est nécessaire. Cette caractéristique a fait le succès de l'approche orientée web dans les entreprises.

1. Diminutif de Web Application.

Le déploiement des logiciels, qui ce soit des primo installations ou des mises à jour, est un aspect coûteux de l'exploitation d'applications en entreprise. Quand il est alors devenu possible d'implémenter des solutions d'entreprise robustes accessibles par un simple navigateur, les éditeurs se sont rués.

Au début, l'ergonomie des premières webapp n'arrivait pas au niveau de leurs équivalents en applications « desktop » ; à chaque envoi d'information au serveur, l'ensemble de l'interface graphique, ici la page HTML, devait se redessiner entièrement. Heureusement la technologie a progressé, notamment grâce à la popularisation d'AJAX (*Asynchronous JavaScript and XML*) qui évite de rafraîchir toute la page à chaque soumission de données.

Le JavaScript de façon générale (langage exécuté localement sur le navigateur) est la pierre angulaire à la création d'une interface homme/machine réactive et intuitive. Au travers de l'API DOM (*Document Object Model*), qui permet de manipuler des éléments de la page HTML, et de la norme à CSS (*Cascading Style Sheets*), langage de description de présentation, il est aujourd'hui possible de créer des composants graphiques web extrêmement poussés n'ayant plus rien à envier ceux des applications classiques.

Comme mentionné plus haut, Android est équipé d'un moteur de rendu HTML et d'un interpréteur JavaScript dernier cri : WebKit. La puissance de WebKit et sa compatibilité avancée avec les spécifications JavaScript et CSS le rend plus qu'apte à servir de client à des applications web de dernières générations. Ainsi, l'option webapp pour un développement mobile Android est un choix tout à fait envisageable et qui peut se révéler très pertinent.

En outre, WebKit étant embarqué dans de nombreux mobiles, une application web ciblant initialement la plateforme Android aura de bonnes chances de fonctionner aussi pour l'iPhone, le Palm Pré ou un smartphone Nokia. Et si ce n'est le cas, il devrait être possible d'y parvenir avec un effort raisonnable d'adaptation. *A contrario*, d'une application native Android qui ne sera pas exécutable sur d'autres environnements.

Si faire un développement web mobile a donc des avantages sur un développement natif, il faut bien comprendre néanmoins que certaines applications, par nature, se prêtent plus à l'orientation web que d'autres. C'est au demeurant la même chose que pour les applications de bureau. En général, les logiciels de gestion, où, de façon schématisée, l'objet du programme est de présenter, créer ou mettre à jour des informations partagées par un ensemble des utilisateurs, collent bien aux technologies web.

À l'opposé, les applications qui font un usage intensif des capacités matérielles du terminal, comme les jeux vidéos par exemple, ne sont pas destinées à fonctionner dans un navigateur. Impossible en effet d'accéder au GPS ou à l'accélération graphique¹ en JavaScript. Enfin, le réseau Internet a beau être omniprésent, l'absolue nécessité de devoir être connecté pour utiliser une webapp peut être une contrainte gênante.

1. Très récemment, Mozilla au travers du Khronos Group a l'ambition d'offrir des fonctionnalités Open GL aux moteurs JavaScript.

Ce qu'il faut retenir

Par bien des aspects, le développement d'une application Android peut s'apparenter à un développement Java classique. Tout est d'ailleurs fait pour donner cette impression : les classes utilisées sont celles du JDK, les outils sont les mêmes (Eclipse, JUnit, Ant...).

Il ne faut cependant pas se méprendre : une application Android est destinée à tourner sur un appareil aux capacités réduites et où les anomalies de fonctionnement peuvent avoir des conséquences plus importantes que sur un poste de travail lambda. Il est préférable de bien garder en tête cette caractéristique.

4

Construire l'interface graphique

Objectifs

Ce chapitre est le plus concret des chapitres de ce livre, il traite de l'interface graphique qui servira au dialogue homme/machine.

La plupart des composants IHM ou widgets sont des classiques déjà utilisés depuis plus d'une décennie sur les applications de bureau. Néanmoins, contrairement aux tout premiers systèmes mobiles d'il y a quelques années maintenant, qui se contentaient de reproduire en miniature les éléments d'affichage traditionnels, les composants Android ont véritablement un look et une ergonomie pensés pour les terminaux tactiles à taille réduite.

4.1 LE RENDU VISUEL

Les objets constituant l'interface graphique se ventilent en deux catégories : les objets héritant de la classe *android.view.View* et ceux de la classe *android.view.ViewGroup*. À l'instar du toolkit AWT où la classe *java.awt.Container* étend la classe *java.awt.Component*, la classe *ViewGroup* étend elle aussi la classe *View*.

Les objets de type *View* sont les widgets, c'est-à-dire les composants gérant les interactions utilisateur. Les widgets ont dans leur très grande majorité une représentation visuelle.

Les *ViewGroup*, quant à eux, offrent rarement un rendu graphique d'eux-mêmes, leur rôle étant avant tout de regrouper ensemble des widgets. La logique est la même que de nombreux frameworks graphiques comme par exemple Java Swing. Sur Android, il n'y a cependant pas de séparation entre le composant parent destiné à contenir les éléments élémentaires de l'interface graphique (dans Swing, on utilise généralement une instance de *JPanel*) et l'objet responsable de la disposition de ces objets. Toujours dans Swing, ce sont les implémentations de *LayoutManager* qui assurent ce travail.

Ici, dans Android, ces deux rôles sont portés par les *ViewGroup* qui sont donc à la fois les objets auxquels sont ajoutés les *View* et les objets gouvernant le rendu graphique de ces éléments.

4.2 APPROCHE PROGRAMMATIQUE OU DÉCLARATIVE

La construction de l'interface homme/machine peut se faire selon deux approches différentes. La première est programmatique, c'est-à-dire que les widgets sont instanciés dans le code à l'aide du mot-clé « *new* » comme n'importe quels autres objets puis ajoutés à une instance de *ViewGroup* en appel de la méthode *addView*. Ce dernier peut lui-même être ajouté à un autre *ViewGroup*, bien entendu tous ces objets autorisent que l'on modifie leurs propriétés d'affichage par le biais de méthodes *ad hoc*. En somme, cette approche correspond exactement à celle employée par les développeurs Java Swing où la définition des écrans est entièrement réalisée dans le code Java.

Le code ci-dessous est l'implémentation de l'activité du projet créé au chapitre précédent. Au démarrage de l'application, la méthode *onCreate* est invoquée.

En paramètre de cette méthode, un objet de type *Bundle* est fourni. Cet objet encapsule l'état de l'activité tel qu'il a pu être sauvegardé précédemment. Un *Bundle* est une sorte de *Map*, bien que *Bundle* n'implémente pas l'interface *java.util.Map*, dont les clés sont des *String* qui référencent des objets *Parcelable* (plus ou moins l'équivalent de *java.io.Serializable*).

Lors du premier démarrage de l'application, le *Bundle* est nul. Il ne sera valorisé que lorsque l'application sera créée à nouveau après avoir été tuée par le système pour libérer de la mémoire.

Cette gestion du cycle de vie de l'application n'est pas sans rappeler le mode de fonctionnement des serveurs Java EE qui ont des mécanismes de passivation et d'activation des *EJB Session Stateful* qui veillent à conserver leur état.

La première instruction, générée par le *plug-in* Eclipse, consiste à appeler la méthode *onCreate* définie au niveau de la classe parent. Cet appel doit être conservé faute de quoi une exception sera lancée.

```
package org.florentgarin.android.demo;
import android.app.Activity;
import android.os.Bundle;
import android.widget.EditText;
import android.widget.LinearLayout;
import android.widget.TextView;
public class DemoActivity extends Activity {
    /*
        * L'interface graphique est définie programmatiquement.
        *
        */
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        LinearLayout layout = new LinearLayout(this);
        layout.setOrientation(LinearLayout.VERTICAL);
        TextView firstNameLabel = new TextView(this);
        firstNameLabel.setText("Prénom");
        EditText firstNameText = new EditText(this);
        TextView lastNameLabel = new TextView(this);
        lastNameLabel.setText("Nom");
        EditText lastNameText = new EditText(this);
        layout.addView(firstNameLabel);
        layout.addView(firstNameText);
        layout.addView(lastNameLabel);
        layout.addView(lastNameText);
        setContentView(layout);
    }
}
```

L'interface graphique est ici créée programmatique. Les widgets utilisés sont des *TextView* et des *EditText*. Le *TextView* est un simple label à la manière du *JLabel* de Swing. L'*EditText* est un champ de saisie de texte sur une ligne unique. Chez Swing, cela correspond au *JTextField*.

Ces composants graphiques sont tous ajoutés à un même *ViewGroup*, en l'occurrence une instance de *LinearLayout*. Sans surprise, ce conteneur gère son layout de façon linéaire. Les composants sont donc alignés les uns à la suite des autres, l'attribut *orientation* sert à déterminer si les widgets seront empilés verticalement ou horizontalement.

Enfin, la dernière commande, *setContentView*, rattache le *ViewGroup* et donc indirectement l'ensemble des widgets à la vue de l'activité.

Il ne reste plus qu'à exécuter un clic droit sur le projet Eclipse puis « Run As/AndroidApplication » et à déclencher le démarrage de l'émulateur et le chargement de l'application.

La fenêtre, ci-dessous apparaît alors :



Figure 4.1 – L'émulateur Android

La création d'interface graphique par le biais de lignes de code n'est pas forcément la meilleure stratégie qui soit. C'était d'ailleurs un reproche fréquemment fait à Swing. L'interface graphique est souvent l'affaire de spécialistes en design ou en ergonomie. Il est avantageux de pouvoir séparer la présentation de la logique applicative pour permettre à ces deux problématiques d'être prises en charge par des équipes dédiées et d'évoluer distinctement l'une de l'autre.

Cette conception plus moderne de la création d'IHM a aujourd'hui fait son chemin et bon nombre de frameworks ont choisi cette voie-là. Flex a son MXML, Mozilla a XUL, Silverlight a XAML. JavaFx aussi offre cette séparation en s'appuyant sur un DSL (*Domain-Specific Language*) dont le domaine est l'interface graphique. Dans ce cas, le langage de description n'est pas à base d'XML mais le principe est le même.

De plus, dans Android comme souvent d'ailleurs, l'interface graphique est structurellement hiérarchique. Les instances de *View* sont regroupées en *ViewGroup* sur plusieurs niveaux pour former un arbre d'objets. On s'aperçoit bien qu'une telle arborescence colle parfaitement à la nature du format xml.

Les développeurs d'Android ont eu finalement la bonne idée de faire cohabiter ces deux modes de création d'écran. L'exemple ci-après reprend la même IHM que précédemment mais fabriquée à l'aide d'un fichier déclaratif xml.

Le fichier .java de l'activité est dorénavant celui-ci :

```
package org.florentgarin.android.demo;
import android.app.Activity;
import android.os.Bundle;
public class DemoActivity extends Activity {
    /*
     * L'interface graphique est définie déclarativement
     * par le fichier "res/layout/main.xml".
     */
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

Les instanciations des widgets et du Layout ont complètement disparu. La méthode onCreate se termine bien par l'appel à la méthode setContentView mais cette fois-ci ce n'est pas la référence de l'objet de type View de premier niveau qui est passé en paramètre mais une constante du type primitif int, issue de la classe R, identifiant le fichier xml de définition de l'interface.

Au niveau du fichier ressource « res/layout/main.xml », le contenu est le suivant :

```
<?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:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Prénom"
        />
    <EditText
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        />
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Nom"
        />
    <EditText
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        />
</LinearLayout>
```

Le composant *ViewGroup LinearLayout* est l'élément racine du fichier xml ce qui semble logique puisque que c'est le composant graphique de plus haut niveau, rattaché à l'activité par la méthode *setContentView*. Le tag xml possède bien un attribut *android:orientation* dont la valeur est *vertical*.

Sous cet élément se retrouvent bien les quatre widgets, deux *TextView* et deux *EditText*. À la manière d'une page HTML, l'ordre dans lequel ils apparaissent a bien sûr son importance ; les éléments étant ajoutés verticalement les uns après les autres en commençant par le haut.

Deux nouveaux attributs ont fait leur apparition : *android:layout_width* et *android:layout_height*. Leurs valeurs respectives sont *fill_parent* et *wrap_content*. Dans la version programmatique de l'exemple, ils n'étaient pas spécifiés car ce sont les valeurs par défaut. Néanmoins, il est impératif de les mentionner dans le fichier xml sinon l'application ne démarrera pas. Dans le code, cela aurait donné ceci :

```
LinearLayout.LayoutParams params = new LinearLayout.LayoutParams(  
    LinearLayout.LayoutParams.FILL_PARENT,  
    LinearLayout.LayoutParams.WRAP_CONTENT);  
firstNameLabel.setLayoutParams(params);
```

Ces attributs xml correspondent donc à la définition de l'objet *LayoutParams*. Le *LayoutParams* est un objet portant des informations sur la politique de placement et de dimensionnement souhaité par le widget. Le terme « souhaité » a son importance, car même si le client par le biais de cet objet peut communiquer à son *ViewGroup* parent ses préférences de rendu visuel, c'est néanmoins ce dernier qui décide du positionnement et de la taille exacte des *View* qu'il contient. Pour les familiers de Swing, le *LayoutParams* peut leur rappeler les objets « constraints » comme le *GridBagConstraints*.

Plus précisément, dans l'exemple ci-dessus, le widget *firstNameLabel* indique qu'au niveau de la largeur « width », par la constante *LinearLayout.LayoutParams.FILL_PARENT*, il aimerait être agrandi au maximum de la place dont dispose son contenant (le *ViewGroup* parent).

Par contre, sur l'axe de la hauteur « height », il préfère simplement avoir la taille nécessaire pour afficher son propre contenu.

Il existe en fait toute une hiérarchie de *LayoutParams*, chaque objet *Layout* (*ViewGroup*) possède presque systématiquement son propre objet *LayoutParams*, héritant de *ViewGroup.LayoutParams*, définissant des attributs adaptés au *ViewGroup* en question. À la manière du *LinearLayout.LayoutParams*, ces objets sont des classes internes static du *Layout*. Ce type de pattern est assez fréquent sur Android, il a le mérite de bien regrouper des objets conçus pour fonctionner ensemble !

Ce système de positionnement des composants graphiques peut paraître fastidieux à tous ceux étant habitués à la création d'IHM sous Visual Basic où les fenêtres se construisent, plaçant des widgets sur une grille au pixel près. Cependant, c'est en s'efforçant de ne jamais recourir au positionnement absolu qu'on obtient une interface graphique parfaitement indépendante de la résolution physique du matériel chargé de l'afficher. Ce principe peut d'ailleurs s'appliquer aux IHM web définies sous CSS.

4.3 LES COMPOSANTS GRAPHIQUES

La brique élémentaire de construction des interfaces graphiques est donc le widget. Ce chapitre passe en revue ceux inclus de base dans le SDK.

Les composants sont créés ici au travers du fichier déclaratif xml. Cette méthode est vraisemblablement préférable, du moins lorsque l'interface graphique est figée, connue à l'avance. Si elle est construite dynamiquement ou doit évoluer au cours de l'exécution du programme, cela se fera bien entendu par le code Java. Il n'est pas antinomique d'utiliser conjointement les deux techniques.

Pour récupérer la référence d'un widget créé depuis le fichier xml de layout, il convient d'utiliser la méthode *findViewById* de la classe *Activity*.

Exemple :

```
Spinner sp = (Spinner)findViewById(R.id.saisons);
```

On peut remarquer que cette méthode, qui n'est pas sans rappeler le *getElementById* de l'objet JavaScript document, accepte en paramètre un int et non un String comme on pourrait s'y attendre. En effet, l'id est exprimé sous forme de constante int, on ne passe pas à la méthode la chaîne de caractères proprement dite. Grâce à cela, la méthode est sûre et on évite ainsi les erreurs à l'exécution qu'on pourrait avoir si on spécifiait un id ne correspondant à aucun widget.

Là où la magie opère, c'est que la liste de ces constantes et plus généralement la classe R tout entière est automatiquement maintenue par le *plug-in* Eclipse. Dès qu'un widget est ajouté dans le fichier xml et que son id est renseigné par l'attribut xml `android:id`, une nouvelle constante apparaît dans la classe R !

4.3.1 TextView

Le *TextView* est le widget le plus basique qu'il soit : une simple zone de texte ! Évidemment la classe définit de nombreux attributs Java et ses équivalents XML (dans sa version déclarative tag XML) pour gouverner finement sa représentation (couleur, police de caractères, dimensions...) et son comportement (conversion automatique des adresses mail, numéros de téléphone et liens web en éléments cliquables...).

L'exemple ci-dessous montre quatre *TextView* positionnés verticalement.

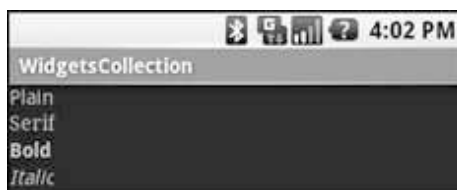


Figure 4.2 — Plusieurs TextView

TagXML

```
<?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:text="Plain" android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <TextView android:text="Serif" android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:typeface="serif" />
    <TextView android:text="Bold" android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:textStyle="bold" />
    <TextView android:text="Italic" android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:textStyle="italic"
    />
</LinearLayout>
```

Deux attributs intéressants ont été introduits dans le fichier de définition du layout. Ce sont les attributs « `android:textStyle` » et « `android:typeface` ». Le premier accepte les valeurs « `normal` », « `bold` » et « `italic` » et le second « `normal` », « `sans` », « `serif` » et « `monospace` ».

Les valeurs peuvent se cumuler grâce au « `|` ». Ainsi si l'on affecte à l'attribut `textStyle` la valeur « `italic|bold` », le texte apparaîtra à la fois en italique et en gras.

4.3.2 EditText

L'*EditText* est une extension du *TextView*. Sans surprise, ce widget est un champ de texte éditable.

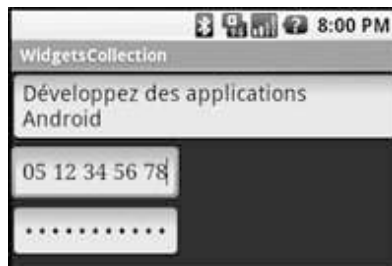


Figure 4.3 — Quelques EditText

Des masques de saisie peuvent être rattachés à l'*EditText*. Dans l'exemple, l'attribut `android:phoneNumber="true"` filtre les caractères qui pourront être saisis (aucune lettre ne pourra être entrée) aiguillant ainsi les utilisateurs. D'autres filtres sont nativement supportés comme `android:capitalize` qui oblige une saisie en majuscule de toutes les lettres, la première lettre du mot ou celle de la phrase selon configuration. L'attribut `android:digits` quant à lui liste purement et simplement les caractères autorisés dans la zone.

Des effets de transformation peuvent également s'appliquer, `android:password="true"` camoufle le mot de passe saisi en remplaçant les caractères par des '*' au niveau de l'affichage.

Si les possibilités de l'*EditText* accessibles par paramétrage XML ne suffisent pas, la classe possède une API Java où il est possible de référencer des implémentations de *KeyListener* grâce auxquels les règles de filtrage et de transformation les plus spécifiques pourront être codées.

TagXML

```
<?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">
    <EditText android:id="@+id/title" android:text="Développez des
    applications Android"
        android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
    <EditText android:id="@+id/phoneNumber" android:text="05 12 34 56 78"
        android:layout_width="wrap_content"
    android:layout_height="wrap_content"
        android:typeface="serif" android:phoneNumber="true" />
    <EditText android:id="@+id/password" android:text="monPassword"
        android:layout_width="wrap_content"
    android:layout_height="wrap_content"
        android:password="true" />
</LinearLayout>
```

4.3.3 CheckBox

La classe *CheckBox* est une case à cocher identique au tag `<input type="checkbox"/>` des formulaires HTML.



Figure 4.4 — Des CheckBox

Comme pour l'*EditText* ou le *TextView*, un style peut être appliqué sur le label des « checkbox ».

Ensuite, dans le code, pour récupérer la valeur de la case à cocher, il faut faire ainsi :

```
CheckBox cb = (CheckBox) findViewById(R.id.italic_cb);  
boolean checked = cb.isChecked();
```

Tag 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">  
    <CheckBox android:id="@+id/plain_cb" android:text="Plain"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content" />  
    <CheckBox android:id="@+id/serif_cb" android:text="Serif"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:typeface="serif" />  
    <CheckBox android:id="@+id/bold_cb" android:text="Bold"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:textStyle="bold" />  
    <CheckBox android:id="@+id/italic_cb" android:text="Italic"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:textStyle="italic" />  
</LinearLayout>
```

4.3.4 ToggleButton

Le *ToggleButton* est un bouton poussoir qui, à l'instar des cases à cocher, a deux états. Cependant, sur le plan de l'ergonomie, la finalité n'est pas la même ; le *ToggleButton* possède deux attributs textuels, un affiché lorsque la valeur du composant est à vrai et un autre lorsque celui-ci est à faux. Un *ToggleButton* est donc plus approprié qu'une *CheckBox* dans les cas où les deux états ne s'expriment pas aussi simplement que par un texte unique associé à une valeur booléenne.

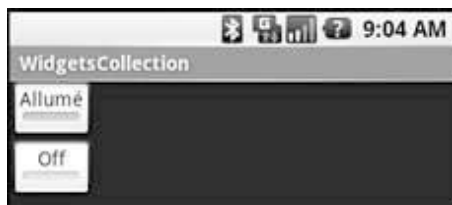


Figure 4.5 — Les deux états du ToggleButton

Tag 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">
    <ToggleButton android:id="@+id/toggle1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textOn="Allumé" android:textOff="Eteint" />
    <ToggleButton android:id="@+id/toggle2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textOn="On" android:textOff="Off" />
</LinearLayout>
```

4.3.5 RadioGroup

La classe *RadioGroup*, accompagnée de la classe *RadioButton*, sert à afficher des boutons radio. Les boutons radio définissent un ensemble d'options parmi lesquelles l'utilisateur ne pourra en choisir qu'une.

En HTML, il n'y a qu'un seul tag « `input type="radio"` » pour définir cet ensemble. Le regroupement des boutons radio se fait par le nom qui est identique :

```
<input type="radio" name="color" value="Noir"/>
<input type="radio" name="color" value="Rouge"/>
<input type="radio" name="color" value="Jaune"/>
<input type="radio" name="color" value="Vert"/>
```

Sur Android, il y a deux classes distinctes : la classe *RadioButton* représente une option sélectionnable et la classe *RadioGroup* est le container (*ViewGroup*) qui rassemble ces options.

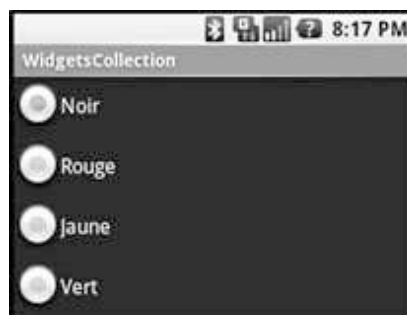


Figure 4.6 — Quatre *RadioButton*

Ainsi, lorsque l'utilisateur clique sur un bouton radio, celui-ci se retrouve sélectionné et si un autre bouton radio appartenant au même groupe est sélectionné, il est désélectionné aussitôt.

TagXML

```
<?xml version="1.0" encoding="utf-8"?>
<RadioGroup android:id="@+id/color" android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:checkedButton="@+id/option1">
    <RadioButton android:id="@+id/option1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:text="Noir"
        android:checked="false" android:layout_gravity="left">
    </RadioButton>
    <RadioButton android:id="@+id/option2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:text="Rouge"
        android:checked="false" android:layout_gravity="left">
    </RadioButton>
    <RadioButton android:id="@+id/option3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:text="Jaune"
        android:checked="false" android:layout_gravity="left">
    </RadioButton>
    <RadioButton android:id="@+id/option4"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:text="Vert"
        android:checked="false" android:layout_gravity="left">
    </RadioButton>
</RadioGroup>
```

À la lecture de ce fichier xml, un œil aiguisé s'arrêterait sur une petite subtilité : au niveau du tag du *RadioGroup* se trouve l'attribut `android:checkedButton="@+id/option1"`.

Cet attribut fixe l'option1 comme étant l'option sélectionnée par défaut. Ce qui peut étonner c'est le signe « + » placé devant id. Le « + » signifie qu'il s'agit d'une création d'identifiant. On voit donc que la création d'un identifiant doit intervenir la première fois où celui-ci est utilisé dans le fichier xml et cela, même si ce n'est pas à l'endroit où il est précisément affecté à un composant. En effet, le premier *RadioButton*, contrairement aux autres, définit son id sans le « + » : `android:id="@+id/option1"`.

4.3.6 Button

La classe *Button* permet de créer un bouton qui pourra être actionné par l'utilisateur pour déclencher un traitement donné.

Il est intéressant de remarquer que la classe *Button* hérite de la classe *TextView* comme d'ailleurs quelques autres widgets comme l'*EditText*. C'est de cet héritage que viennent les capacités des boutons à pouvoir appliquer des styles à leur label.



Figure 4.7 — De simples Button

Pour attacher un écouteur à l'événement généré par le bouton quand ce dernier est cliqué, il faut procéder ainsi :

```
final Button button = (Button) findViewById(R.id.okButton);
button.setOnClickListener(new Button.OnClickListener() {
    public void onClick(View v) {
        // action à exécuter lorsque le bouton
        // est pressé
    }
});
```

Cela ressemble aux *ActionListener* de Swing si ce n'est que, comme le nom de la méthode *setOnClickListener* le laisse présager, il n'est pas possible d'ajouter plusieurs écouteurs.

Dans l'exemple, la référence « button » est précédée du modificateur « final » qui signifie en Java que la référence ne pourra pas pointer vers un autre objet. Dans un environnement contraint aux capacités matérielles limitées, ce mot-clé, en informant le compilateur que la référence ne changera pas de zone mémoire, lui permet d'effectuer des optimisations qui se ressentiront sur les performances. C'est une bonne pratique de penser à l'utiliser dans les applications Android.

L'interface à implémenter pour coder notre écouteur *Button.OnClickListener* est une interface publique et statique définie en réalité au niveau de la classe *View*. Par héritage, *Button.OnClickListener* fait donc référence à l'interface *View.OnClickListener*. Cette interface aura pu être définie dans son propre fichier de stockage .java ; mais l'usage des types internes est manifestement un trait de caractère de l'API Android !

Tag XML

```
<?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/okButton" android:text="OK"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <Button android:id="@+id/CancelButton" android:text="Annuler"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

4.3.7 Spinner

La classe *Spinner* présente à l'utilisateur une liste de valeurs parmi lesquelles il ne peut en choisir qu'une. Une fois, cette valeur sélectionnée, seule celle-ci reste visible, la liste entière disparaît.

Le *Spinner* se comporte en fait comme ce qui est communément appelé « combo-box » ; toutefois le *Spinner* n'autorise pas la saisie d'une valeur autre qu'une de celles présentes dans la liste.



Figure 4.8 — Sous Android une liste d'options est appelée Spinner

L'attribut « prompt » sert à placer un message donnant des indications sur le choix à opérer.

TagXML

```
<?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">
    <Spinner android:id="@+id/saisons" android:layout_width="wrap_content"
        android:prompt="@string/prompt"
        android:layout_height="wrap_content"
        android:entries="@array/saisons"/>
</LinearLayout>
```

Le fichier de ressource arrays :

```
<resources>
<array name="saisons">
<item>printemps</item>
<item>été</item>
<item>automne</item>
<item>hiver</item>
</array>
</resources>
```


4.3.8 AutoCompleteTextView

L'*AutoCompleteTextView* correspond un peu à un combobox éditable, du moins à ceci près que la liste de choix possibles n'apparaît que lorsque l'utilisateur entre *n* lettres, *n* étant paramétrable grâce à l'attribut `android:completionThreshold`. Bien sûr, il est permis de saisir quelque chose non présent dans la liste.

On peut également placer une petite phrase d'explication en bas de la liste.



Figure 4.9 — L'*AutoCompleteTextView* aide à la saisie

La construction de la liste de valeurs ne peut pas se faire par le fichier de layout xml, il faut le faire par le code Java.

```
AutoCompleteTextView textView = (AutoCompleteTextView)
findViewById(R.id.voiture);
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
android.R.layout.simple_dropdown_item_1line, CARS);
textView.setAdapter(adapter);
```

La première étape consiste à récupérer une référence vers le widget déclaré dans le fichier xml, c'est du classique.

Ensuite, il faut instancier une classe de type *ListAdapter* qui fera le pont entre les données proprement dites et le widget *AutoCompleteTextView*. Dans l'exemple, les données proviennent d'un simple tableau de *String* ; dans la réalité, cette liste pourrait être alimentée dynamiquement par une base de données. Les « Adapter » d'Android peuvent faire penser aux *ListModel* de Swing, cependant l'approche ici est moins puriste (plus pragmatique ?) que celle du JDK. Car le modèle (au sens MVC) et le « Render » ne sont pas dissociés, les adaptateurs renvoyant directement (méthode `getView(int position, View convertView, ViewGroup parent)`) le widget à afficher à la position donnée.

Pour revenir à l'exemple, l'implémentation de *ListAdapter* utilisé est l'*ArrayAdapter*. Son constructeur prend en paramètre le contexte (la classe *Activity* courante), une référence vers un layout qui servira de modèle pour représenter

chaque élément de la liste et enfin le tableau des données. Ici c'est le layout `android.R.layout.simple_dropdown_item_1line` qui est utilisé mais il est tout à fait possible d'utiliser son propre layout. Ce dernier ne devra par contre contenir qu'un unique `TextView` qui accueillera la valeur à afficher. Si l'on souhaite utiliser un layout composé de plus d'un `TextView`, il convient d'indiquer l'id de celui qui affichera la valeur (`ArrayAdapter` offre le constructeur adéquat).

TagXML

```
<?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:text="Quelle est votre voiture ?"
    android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <AutoCompleteTextView android:id="@+id/voiture"
        android:layout_width="fill_parent"
    android:layout_height="wrap_content"
        android:completionHint="Choisissez dans la liste ou saisissez la
    valeur"
        android:completionThreshold="1" />
</LinearLayout>
```

4.3.9 DatePicker

La classe `DatePicker` offre un widget facilitant la saisie de date.



Figure 4.10 — Le widget DatePicker

L'usage de cette classe ne recèle aucune subtilité particulière, les méthodes pour récupérer, de façon aisée, la date spécifiée ou pour la fixer sont bien présentes.

La classe `DatePicker.OnDateChangeListener` permet d'enregistrer un callback pour capter les changements de valeur.

Enfin, la classe utilitaire `DatePickerDialog` affiche une boîte de dialogue contenant uniquement un `DatePicker`. Cette fenêtre toute faite est très pratique quand on veut isoler la saisie de la date d'éventuelles autres informations.

Tag 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">
    <DatePicker android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
</LinearLayout>
```

4.3.10 TimePicker

Le *TimePicker* est le petit frère du widget *DatePicker* mais appliqué à l'heure. Ce widget propose tout ce qu'on peut attendre, c'est-à-dire la saisie par le clavier et le pointeur, un mode AM/PM et 24 heures...

La classe *TimePicker.OnTimeChangedListener* permet d'écouter les événements et *TimePickerDialog* ouvre une fenêtre dévolue à la saisie d'une donnée horaire.

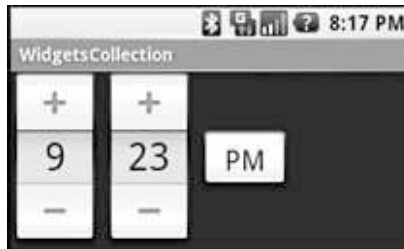


Figure 4.11 — Le TimePicker

TagXML

```
<?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">
    <TimePicker android:id="@+id/time"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

4.3.11 Gallery

Cette classe sert à construire une véritable galerie d'objet *View* défilant horizontalement, l'élément sélectionné restant au milieu.

L'exemple ci-dessous illustre ce composant en présentant une série d'images (des photos de Romain Guy pour être plus exact !) ; c'est l'usage auquel on pense

immédiatement pour ce widget, toutefois, la classe *Gallery* peut être composée d'items de type *View* quelconque pas uniquement d'*ImageView*.

Pour implémenter le mur d'image, il faut alimenter la classe *Gallery* avec une instance d'*Adapter*. C'est le même principe que celui mis en place pour le widget *AutoCompleteTextView*.

On peut légèrement regretter qu'Android ne propose pas d'*Adapter* haut niveau prêt à l'emploi pour réaliser de beaux « coverflows ». Les implémentations d'*Adapter* d'Android sont un peu rudimentaires, il va falloir travailler un peu !

Le code ci-dessous utilise donc sa propre implémentation d'*Adapter*, la classe *ImageAdapter*.

Cette classe est statique et interne à la classe *GalleryActivity*. Cependant, il pourrait être intéressant de la rendre générique et facilement réutilisable en en faisant une classe autonome en la sortant de *GalleryActivity*. Les images sont des ressources qui font partie intégrante de l'application. Elles sont stockées dans le répertoire « res/drawable ».

La classe *ImageAdapter* ne part pas de zéro mais étend la classe abstraite *BaseAdapter*, seules les méthodes essentielles sont donc implémentées dans *ImageAdapter* :

- **getCount()** renvoie au nombre d'éléments de la galerie. L'implémentation va de soi et renvoie simplement la taille du tableau, de type `int`, des références des images.
- **getItem(int position)** doit renvoyer la donnée associée à la position passée en paramètre. Dans le cas présent, il s'agit simplement de la position.
- **getItemId(int position)** donne l'Id de l'élément placé à la position donnée. Dans l'exemple, les images n'ayant pas vraiment d'identifiant, le choix a été fait de renvoyer cette même position.
- **getView(int position, View convertView, ViewGroup parent)** est sans doute la méthode la plus importante car de son implémentation dépend le rendu visuel du widget. La responsabilité de cette méthode est de délivrer l'objet *View* chargé de la représentation de l'élément qui occupe la position fournie en paramètre. La méthode reçoit également deux autres objets en paramètres : « *convertView* » qui est l'ancien objet *View* (il est parfois plus commode de modifier l'ancien *View* que d'en créer un autre) et le *ViewGroup* parent. Dans le cas présent, le paramètre « *parent* » est l'objet *Gallery* lui-même tandis que le *convertView* est toujours nul.

Pour la galerie, la *View* instanciée est bien sûr un *ImageView*. Sa source est la référence du fichier jpg, correspondant à la position de l'image, enregistré dans le répertoire des ressources : `res/drawable`. Les images ont toutes la même taille, il n'a donc pas été nécessaire de les redimensionner ; toutefois, si elles ont des dimensions différentes, il est opportun de les mettre toutes à la même taille fixe en faisant ceci :

```
img.setScaleType(ImageView.ScaleType.FIT_XY);  
img.setLayoutParams(new Gallery.LayoutParams(75, 75));
```

Enfin, la dernière instruction, l'appel à la méthode `setBackgroundResource(int resid)` de l'`ImageView`, peut sembler anodine mais elle est capitale. Sans cela la galerie ne s'affiche incorrectement, les images sembleront tronquées. L'explication est à chercher dans la javadoc de la classe `Gallery`. Cet appel « magique », dont la nécessité pour un fonctionnement par défaut peut sembler discutable, a pour objectif d'affecter la ressource de type `Drawable` dont l'id est passé en paramètre à l'arrière-plan de l'`ImageView`. Le `Drawable` en question est issu du thème de l'activité, c'est cette image qui dessine le cadre, le paramétrage par défaut des attributs de la classe `Gallery` a été défini en présumant que les éléments à disposer sur la galerie auront cet arrière-plan.

La récupération de cet id est détaillée au chapitre « Les thèmes ».

Au final, la classe `GalleryActivity` contient le code suivant :

```
public class GalleryActivity extends Activity {
    private final static int[] IMAGES = { R.drawable.gallery1,
                                           R.drawable.gallery2,
                                           R.drawable.gallery3,
                                           R.drawable.gallery4,
                                           R.drawable.gallery5,
                                           R.drawable.gallery6,
                                           R.drawable.gallery7,
                                           R.drawable.gallery8,
                                           R.drawable.gallery9,
                                           R.drawable.gallery10 };
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.gallery);
        Gallery gallery = (Gallery) findViewById(R.id.gallery);
        ImageAdapter adapter = new ImageAdapter(this, IMAGES);
        gallery.setAdapter(adapter);
    }
    public static class ImageAdapter extends BaseAdapter {
        private int[] m_images;
        private Context m_context;
        private int m_itemBackground;
        public ImageAdapter(Context context, int[] images) {
            m_context = context;
            m_images=images;
            TypedArray array =
context.obtainStyledAttributes(R.styleable.Gallery);
            m_itemBackground = array.getResourceId(
R.styleable.Gallery_android_galleryItemBackground, 0);
            array.recycle();
        }
        public int getCount() {
            return m_images.length;
        }
        public Object getItem(int position) {
            return position;
        }
        public long getItemId(int position) {
            return position;
        }
    }
}
```

```

        public View getView(int position, View convertView, ViewGroup
parent) {
            ImageView img = new ImageView(m_context);
            img.setImageResource(m_images[position]);
            //img.setScaleType(ImageView.ScaleType.FIT_XY);
            //img.setLayoutParams(new Gallery.LayoutParams(75, 75));
            img.setBackgroundResource(m_itemBackground);
            return img;
        }
    }
}

```



Figure 4.12 — Le composant Gallery

TagXML

```

<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:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Gallery Widget Android"
        />
    <Gallery android:id="@+id/gallery"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:gravity="bottom"
        />
</LinearLayout>

```

Dans le répertoire « res/values », il faut créer un fichier contenant les attributs. Dans l'exemple, ce fichier se nomme `attrs.xml`, son contenu est :

```

<resources>
    <declare-styleable name="Gallery">
        <attr name="android:galleryItemBackground">
            </attr>
        </declare-styleable>
    </resources>

```

4.3.12 ImageView

L'*ImageView* est un widget dont la représentation est une image. La source de l'image peut provenir du répertoire layout ou être référencée par une URI.

ImageView intègre des fonctionnalités de transformation, la taille de l'image peut être modifiée et différer de la taille native. L'attribut `android:scaleType` permet de choisir l'algorithme avec lequel l'image sera étirée.



Figure 4.13 — Un ImageView

TagXML

```
<?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">
    <ImageView android:id="@+id/image"
        android:src="@drawable/logo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"/>
</LinearLayout>
```

4.3.13 ImageButton

ImageButton est une sous-classe d'*ImageView* destinée à recevoir des interactions utilisateur comme un bouton.

Par exemple, l'enregistrement d'un écouteur sur le clic se fait de la même façon, c'est-à-dire par la méthode `setOnClickListener` (qui est d'ailleurs définie au niveau de la classe parente *View*). L'*ImageButton* apporte en plus la matérialisation visuelle de ces événements ; lors d'un clic, l'image se verra entourée d'un cadre orange.



Figure 4.14 — L'ImageButton est cliquable

TagXML

```
<?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">
    <ImageButton android:id="@+id/image"
        android:src="@drawable/logo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"/>
</LinearLayout>
```

4.3.14 ProgressBar

Le widget *ProgressBar* rend compte de l'avancement d'une tâche qui dure un certain temps. Quand la progression n'est pas mesurable, le widget sert au moins à montrer qu'il y a toujours de l'activité et qu'il faut gentiment patienter.

La barre de progression est très intéressante à étudier car elle met en lumière une difficulté classique des toolkits graphiques et bien connue des *aficionados* de Java AWT et Java Swing qu'est la gestion des *threads*.

À l'instar donc de Java Swing, l'interface graphique d'Android est conçue pour fonctionner avec un unique thread (UI Thread). Seul ce thread est autorisé à modifier l'IHM sans quoi l'application s'expose à de nombreux bugs. Ce thread est celui qui instancie la classe *Activity* et exécute la méthode *onCreate* de cette dernière. Cela veut dire qu'il est permis de modifier sans crainte la couleur d'un bouton ou le texte d'un libellé. Par contre, attention, il ne faut pas bloquer ce thread par l'exécution d'un traitement long car cela l'empêcherait de redessiner l'écran : il en résulterait une impression de « freeze » caractéristique justement des applications ne se souciant guère de cette problématique.

Dans l'exemple, un worker thread est donc créé pour prendre en charge la tâche longue dont il s'agit de mesurer l'avancement :

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.progress_bar);
    final ProgressBar bar = (ProgressBar) findViewById(R.id.progress);
    final Handler handler = new Handler();
    Thread worker = new Thread(new Runnable() {
        public void run() {
            while (progress < 100) {
                try {
                    //simule un traitement long
                    Thread.sleep(200);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                handler.post(new Runnable() {
                    public void run() {
                        bar.setProgress(++progress);
                    }
                });
            }
        }
    });
    worker.start();
}
```



```

        }
    });
}
});
worker.start();
}

```

Le worker effectue sa tâche de façon fractionnée, c'est la condition *sine qua non* à l'usage d'une barre de progression déterminée. Ici l'activité est simulée par un sleep de 200 millisecondes. La deuxième partie du traitement concerne la mise à jour de l'IHM avec le niveau de progression. La méthode `setProgress`, déclenchant des modifications sur l'IHM, doit absolument être appelée depuis le UI Thread. Il faut donc poster les commandes à dérouler, encapsulées dans une instance de `Runnable`, au UI Thread. Dans Java Swing, cela aurait été fait par l'appel à la méthode `EventQueue.invokeLater(Runnable)`. Dans Android, le principe a été quelque peu généralisé.

En effet, la classe `Handler` permet d'envoyer et de traiter des objets `Message` et `Runnable`. Lorsqu'on instancie l'`Handler` en faisant « `final Handler handler = new Handler();` », il est automatiquement branché sur le `MessageQueue` du thread l'ayant créé. Dans ce cas précis, il s'agit du UI Thread puisque c'est bien lui qui exécute la méthode `onCreate`.

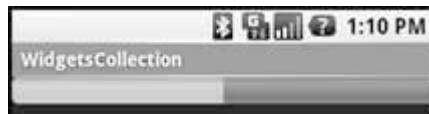


Figure 4.15 — ProgressBar déterminée

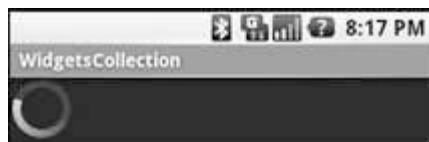


Figure 4.16 — ProgressBar indéterminée

TagXML

```

<?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>

```

L'attribut `style` est obligatoire, sinon la barre de progression affichée sera en mode indéterminée. Dans ce cas, invoquer la méthode `setProgress` n'a aucun effet. L'annotation « ? » est typique des références pointant vers des attributs de thème. Le sujet des thèmes et styles visuels est développé au chapitre intitulé « Styles et thèmes ».

4.3.15 AnalogClock

AnalogClock est une simple horloge analogique qui affiche l'heure courante. Son intérêt est relativement limité car il n'est pas possible de régler l'horloge sur une autre heure.

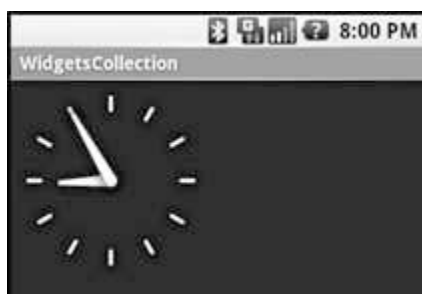


Figure 4.17 — L'AnalogClock dont l'utilité reste à prouver

Tag 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">
    <AnalogClock android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

4.3.16 DigitalClock

DigitalClock est l'équivalent de *AnalogClock* mais le rendu est digital. Ce widget a les mêmes limitations que son frère jumeau *AnalogClock*. Son objectif est clairement d'inscrire à l'écran l'heure système, le mode de l'horloge, AM/PM ou 24 heures, dépendra également de la configuration de l'OS.

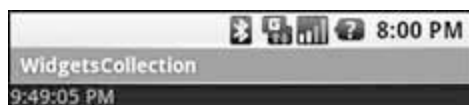


Figure 4.18 — Version digitale de l'horloge

TagXML

```
<?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">
    <DigitalClock android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

4.3.17 RatingBar

Le widget *RatingBar* est apparu récemment avec l'essor des sites sociaux et communautaires. Son rôle est de produire un rendu visuel d'une notation ou, s'il est éditable, de permettre à l'utilisateur d'ajuster cette dernière.

C'est l'attribut `android:isIndicator` qui fixe le caractère éditable du widget.

Les méthodes du composant permettent de fixer l'échelle (nombre d'étoiles), la granularité, ou encore de renseigner la notation par du code.



Figure 4.19 — Un widget très Web 2.0

TagXML

```
<?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">
    <RatingBar android:id="@+id/gallery"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
</LinearLayout>
```

4.4 LES LAYOUTS

Les layouts sont des *ViewGroup* responsables du dimensionnement et du positionnement des widgets à l'écran. Il en existe plusieurs, chacun adoptant une stratégie bien spécifique.

4.4.1 ListView

ListView place les widgets qu'il contient verticalement, les uns après les autres et un seul par ligne : ils sont visuellement séparés par une ligne. Les widgets sont fournis par une instance de *ListAdapter*. Si la hauteur cumulée de tous les composants ajoutés dépasse la taille de l'écran, la liste devient scrollable.

Ce layout pourrait, par exemple, être employé pour créer un menu avec sous-menu : les options du menu seraient des *TextView* qui en cas de sélection afficheraient un autre *ListView*.

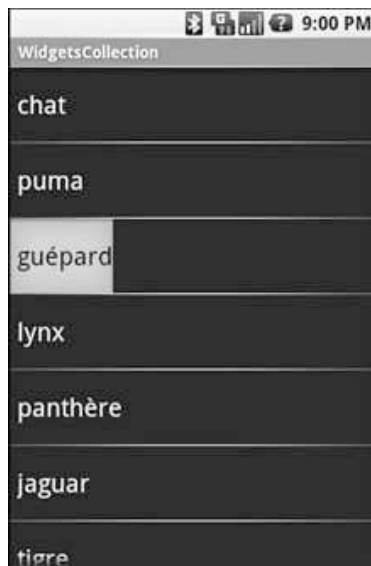


Figure 4.20 — Le ListView peut servir à créer des menus

TagXML

```
<?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">
    <ListView android:id="@+id/felins" android:layout_width="wrap_content"
        android:choiceMode="singleChoice"
        android:layout_height="wrap_content"
        android:entries="@array/felins" />
</LinearLayout>
```

4.4.2 FrameLayout

Ce layout empile les widgets les uns sur les autres. Chaque composant est positionné dans le coin en haut à gauche en masquant le widget précédent à moins que le widget venant d'être ajouté ne soit transparent.

4.4.3 LinearLayout

Ce layout aligne les widgets dans une seule direction : verticalement ou horizontalement selon le paramétrage de l'attribut orientation. Ce layout prend en compte les marges (*margins*) ainsi que l'alignement (*gravity left, center, right*) des composants. Les éléments graphiques peuvent déclarer un poids (*weight*) qui indique comment se répartira le surplus d'espace entre les widgets, l'espace supplémentaire sera distribué proportionnellement aux poids des composants.

Ce layout est fréquemment utilisé, les exemples que l'on peut trouver plus loin dans ce livre l'exploitent pour construire leur IHM.

4.4.4 TableLayout

Le *TableLayout* agence les widgets sur un quadrillage exactement comme on pourrait le faire en HTML avec la balise `<table>` :



Figure 4.21 — Disposition tabulaire

Toutefois, le layout présente quelques différences notables par rapport à son équivalent HTML : il n'est pas possible de faire apparaître les bordures du quadrillage, *TableLayout* ne sert qu'à faire du positionnement d'objets. De plus, une même cellule ne peut pas s'étaler sur plusieurs colonnes.

TagXML

```
<?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="Coord 0.0"
            android:gravity="center"
            android:padding="3dip" />
```

```
<TextView
    android:text="Coord 0.1"
    android:gravity="center"
    android:padding="3dip" />
</TableRow>
<TableRow>
    <TextView
        android:text="Coord 1.0"
        android:gravity="center"
        android:padding="3dip" />
    <TextView
        android:text="Coord 1.1"
        android:gravity="center"
        android:padding="3dip" />
</TableRow>
<TableRow>
    <TextView
        android:text="Coord 2.0"
        android:gravity="center"
        android:padding="3dip" />
    <TextView
        android:text="Coord 2.1"
        android:gravity="center"
        android:padding="3dip" />
</TableRow>
</TableLayout>
```

4.4.5 RelativeLayout

Les widgets contenus dans un *RelativeLayout* peuvent déclarer leur position relativement par rapport à leur parent ou par rapport aux autres widgets. Ce *ViewGroup* est un peu plus difficile à maîtriser que les autres layout car un placement malencontreux peut entraîner en cascade d'autres erreurs.

Par exemple par l'attribut « `android:layout_below="@id/widget2"` », un widget annonce qu'il souhaite se retrouver en dessous du widget2. Avec « `android:layout_toLeftOf="@id/widget3"` », il aurait été mis à la gauche du widget3.

Ce qu'il faut retenir

La création d'interface graphique avec Android peut se faire à la fois par du code et par une définition XML. Cette dernière solution est très souvent souhaitable.

Les familiers d'HTML n'auront pas de mal à se faire à la grammaire XML d'Android de définition des IHM. Les habitués de Swing pourront également déceler des similitudes, néanmoins ceux qui sont réfractaires à ce toolkit n'ont pas à s'inquiéter, l'approche d'Android est plus pragmatique et simple que le framework de Java.

5

Le modèle de composants

Objectifs

Une des caractéristiques d'Android est son approche fortement modulaire qui permet aux applications de collaborer entre elles. Une application est constituée de plusieurs composants, chacun de ces composants pouvant être réutilisé depuis une autre application ce qui évite ainsi de devoir réinventer la roue.

Ce chapitre traite de ce modèle d'architecture, les différents types de composants, la façon de les invoquer et leur cycle de vie sont passés en revue.

5.1 UNE FORTE MODULARITÉ

Il y a plus d'une dizaine d'années, l'avènement des langages objets a fait naître l'espoir de pouvoir développer facilement des éléments de code réutilisables d'une application à l'autre. Malheureusement, si le paradigme objet, avec des notions telles que l'encapsulation, favorise la création de bibliothèques, il n'en demeure pas moins que développer une portion de code réellement exploitable dans un contexte autre que celui d'origine n'est toujours pas chose aisée.

Pour faciliter la collaboration entre les éléments de programme, le modèle applicatif d'Android définit la notion de composants d'applications. Un composant est un module de l'application qui pourra être activé depuis un autre programme Android. Lorsqu'un module est activé, l'application à laquelle il appartient est lancée si nécessaire, il est donc exécuté dans son propre processus.

Les fonctionnalités du module sont donc réutilisées en mettant en œuvre des communications interprocessus.

5.2 QUATRE FAMILLES DE COMPOSANTS

Android prévoit quatre catégories de composants :

- les objets de la classe *Activity*,
- les objets de la classe *Service*,
- les objets de la classe *BroadcastReceiver*,
- et enfin les objets du type *ContentProvider*.

Les objets de ces types sont donc des morceaux de l'application qui pourront être invoqués par d'autres applications pour remplir une tâche précise. C'est un peu comme si le programme avait un point d'entrée officiel, celui de l'activité principale qui se lance lorsque l'utilisateur clique sur l'icône de l'application, et des points d'entrée alternatifs démarrés par d'autres programmes.

5.3 LA NAVIGATION ENTRE ACTIVITÉS

Les activités sont des composants centraux des applications. Ce sont également les composants les plus remarquables car ils portent les éléments visuels de l'interface utilisateur agencés sur l'écran.

La navigation entre les écrans peut se faire de deux façons différentes : soit explicitement, soit implicitement. Dans les deux cas, l'ordre de changement d'activité est véhiculé par un objet de type *Intent* (intention en anglais).

Dans le mode explicite, les activités s'enchaînent les unes aux autres par invocation directe. C'est-à-dire qu'une activité donnée déclenche l'affichage d'une autre activité en appelant la méthode *startActivity* avec un *Intent* mentionnant clairement le nom de l'activité. On verra par la suite comment créer concrètement des objets *Intent*.

Le mode explicite est donc très classique : comme dans la plupart des applications, les écrans à afficher sont invariablement les mêmes d'une exécution à l'autre et identifiés à l'avance. Aucune particularité n'est à noter dans ce mode de fonctionnement si ce n'est qu'Android permet d'afficher des activités n'appartenant pas à l'application d'origine.

Le mode implicite par contre est une spécificité d'Android extrêmement puissante. Dans ce mode, le nom de l'activité n'est pas précisé nominativement. L'objet *Intent* qui encapsule la demande de changement ne porte pas l'identification de l'activité mais un descriptif des caractéristiques ou plutôt des capacités de traitement dont l'activité devra être dotée. Ensuite, une mécanique de résolution se met en marche, Android recherche dans tout le système, et non pas uniquement dans l'application courante, les activités répondant aux exigences exprimées dans l'*Intent*. À la fin de cet algorithme, l'activité identifiée s'affiche alors. Au cas où il y aurait plusieurs activités en mesure d'assurer le service demandé, leur liste est proposée à l'utilisateur qui devra alors en sélectionner une. Il peut arriver aussi qu'aucune activité ne puisse couvrir le besoin. Si cela se produit, une exception est alors lancée.

5.3.1 L'objet Intent

L'objet au cœur du dispositif de la navigation des écrans est l'*Intent* qui informe des « intentions » de l'activité sur le traitement suivant à réaliser. Un *Intent* est composé de trois attributs essentiels qui participent à l'identification de l'activité à afficher :

- Action
- Data/Type
- Category

L'action est une chaîne de caractères qui symbolise le traitement à déclencher. De nombreuses constantes sont définies dans le SDK pour les actions nativement disponibles. Parmi elles, on peut citer à titre d'exemple `Intent.ACTION_WEB_SEARCH` pour demander de réaliser une recherche sur Internet ou encore `Intent.ACTION_CALL` pour passer un appel téléphonique.

L'attribut Data est une donnée qui détaille l'action de l'*Intent* dont elle dépend directement. Elle peut être d'ailleurs nulle, certaines actions n'appelant pas forcément à devoir être précisées. Pour illustration, dans l'exemple `ACTION_CALL`, l'attribut Data sera le numéro de téléphone à composer.

L'attribut Data est couplé avec un autre attribut qui est le type MIME de la donnée à traiter. En principe, on spécifie soit le type soit la « data ». L'appel à la méthode `setType(String)` efface l'attribut data qui aurait pu être renseigné par la méthode `setData(Uri)` et réciproquement. La raison de cela est que dans la majorité des cas, le type MIME peut être déterminé en fonction du data. Si on souhaite malgré tout explicitement fixer le type MIME de la donnée, il faut utiliser la méthode `setDataAndType(Uri,String)`.

On pourrait se demander pourquoi renseigner le type si on ne fournit pas de donnée. La réponse est que l'activité appelée et appelante communiquent dans les deux sens. L'intention (*Intent*) est créée par l'activité de départ et transmise à la suivante. Celle-ci a aussi la possibilité de retourner des données, toujours transportées par un objet de type *Intent*, à la première activité. On peut donc imaginer que l'*Intent* initial ne contienne pas de data mais seulement le type MIME souhaité pour le format de la réponse.

La catégorie, quant à elle, apporte une classification à l'action. La catégorie `Intent.CATEGORY_LAUNCHER` positionne l'activité comme étant exécutable, cette catégorie est utilisée de pair avec l'action `Intent.ACTION_MAIN`. Android positionne les activités de cette catégorie dans le lanceur d'applications. La catégorie `CATEGORY_HOME` marque l'activité à afficher au démarrage du téléphone.

L'objet *Intent* possède également d'autres attributs tels que les flags et les extras. Ces attributs sont d'autres possibilités de transmission d'informations à l'activité appelée. Ces attributs sont néanmoins accessoires dans la mesure où ils n'entrent pas en compte dans le processus de recherche de l'activité cible.

5.3.2 La résolution

La résolution est le mécanisme de détermination de la ou des activités aptes à gérer l'action exprimée par l'intention. L'algorithme repose sur la confrontation de l'objet *Intent* et les *IntentFilter* des activités présentes sur le système Android.

Un *IntentFilter* est un objet rattaché à une activité par lequel cette dernière informe publiquement de ses capacités. Cette déclaration est réalisée comme on peut s'en douter dans le fichier manifeste.

Une activité peut définir un ou plusieurs *IntentFilter*, chacun étant une fonction que l'activité peut remplir. Pour être sélectionnée, une activité devra avoir un *IntentFilter* remplissant à lui seul entièrement le contrat matérialisé par l'*Intent*.

L'extrait suivant du manifeste déclare un unique *IntentFilter* pour l'activité *IntentTesterActivity*. Celui-ci possède trois actions, une catégorie et une entrée data avec le type MIME d'indiqué :

```
<activity android:name=".IntentTesterActivity"
  android:label="@string/app_name">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <action android:name="android.intent.action.VIEW"/>
    <action android:name="android.intent.action.DELETE"/>
    <category android:name="android.intent.category.LAUNCHER" />
    <data android:mimeType="vidéo/mpeg"/>
  </intent-filter>
</activity>
```

La résolution des intentions peut se schématiser comme on le voit sur la figure 5.1.

En détail, l'algorithme compare un à un les trois attributs majeurs des intentions et des *IntentFilters* que sont les actions, les data/type et les catégories. Si la comparaison échoue sur un de ces points, la fonctionnalité représentée par l'*IntentFilter* sera écartée pour incompatibilité.

Pour être valide par rapport à l'*Intent*, l'*IntentFilter* devra avoir dans sa liste des actions celle qui est spécifiée dans l'*Intent*. Si l'*Intent* n'en mentionne aucune, ce premier test sera validé à condition que l'*IntentFilter* ait au moins une action.

Vient ensuite le test des catégories. Toutes les catégories référencées dans l'*Intent* devront être présentes dans la déclaration de l'*IntentFilter*. Bien sûr, l'*IntentFilter* pourra en définir d'autres encore ; qui peut le plus peut le moins. Toutefois, les *IntentFilter* devront en plus mentionner la catégorie *Intent.CATEGORY_DEFAULT* car les activités lancées par la méthode *Context.startActivity* sont tenues d'avoir cette catégorie.

Enfin, le test portera sur la nature des données. La logique est toujours la même : l'URI et/ou le type MIME de l'*Intent* doit figurer dans la liste des objets traitables par l'activité. L'*IntentFilter* peut recourir à l'usage de « wildcards ». Ainsi dans l'exemple de déclaration précédent, le type MIME supporté par l'activité est « vidéo/mpeg ». Cela veut dire qu'un *Intent* référençant une vidéo au format mov, dont le type MIME est « vidéo/quicktime » ne pourra pas être passé à cette activité.

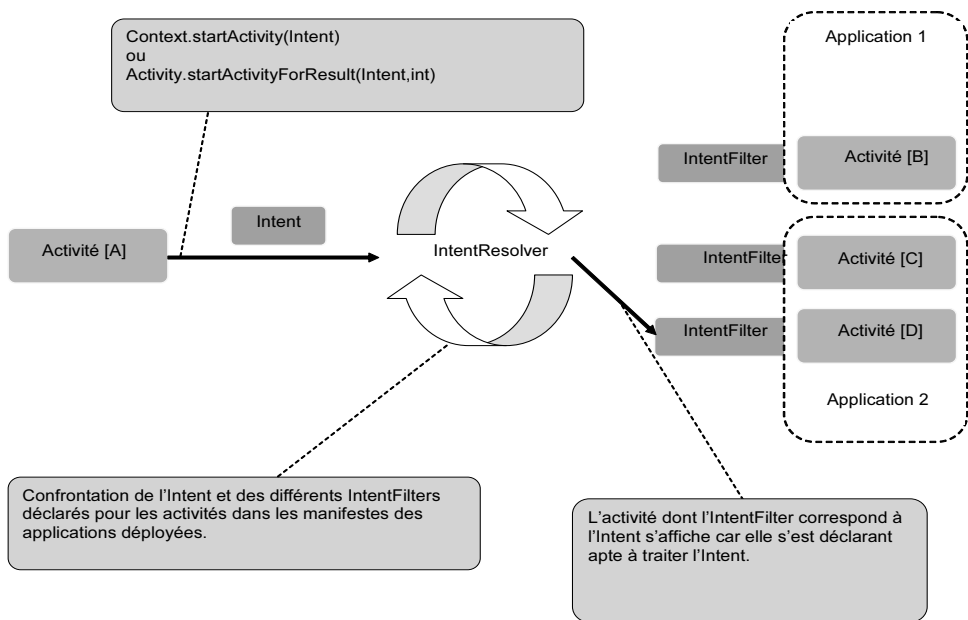


Figure 5.1 — - Processus de résolution des Intents

Par contre, si l'*IntentFilter* avait déclaré comme type MIME « vidéo/* », cela aurait convenu. À l'instar du type, l'URI peut n'être définie qu'en partie au niveau du filtre :

```
<data android:mimeType="vidéo/*" android:scheme="http"/>
```

Cette déclaration accepte toutes les vidéos accessibles par http.

```
<data android:mimeType="vidéo/*" android:scheme="http"
android:host="www.youtube.com"/>
```

Et celle-ci toutes les vidéos délivrées par youtube. Plus l'URI est précisée finement dans l'*IntentFilter*, plus la contrainte de sélection sera forte.

Ci-dessous, un exemple de code déclenchant l'affichage du dialer avec un numéro prérempli :

```
private void startDialActivity(){
    Intent dial = new Intent();
    dial.setAction(Intent.ACTION_DIAL);
    dial.setData(Uri.parse("tel:1234567"));
    startActivity(dial);
}
```

Le dialer est une activité native d'Android, l'exemple montre clairement que les activités peuvent s'enchaîner sans devoir nécessairement appartenir à la même application. Du point de vue système bas niveau, il est à souligner que l'activité dialer tourne dans son propre processus et non pas dans celui de l'activité appelante.

Si on s'amuse à créer une activité réagissant à l'action `Intent.ACTION_DIAL` et à l'enregistrer tel quel dans le système par la définition d'un *IntentFilter* ad hoc :

```
<activity android:name=".FakeActivity">
    <intent-filter>
        <action android:name="android.intent.action.DIAL" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:scheme="tel" />
    </intent-filter>
</activity>
```

Alors, à l'exécution du code précédent, comme deux activités présentes sur Android seront capables de recevoir l'objet *Intent*, une boîte de dialogue demandant à l'utilisateur de trancher apparaîtra :



Figure 5.2 — L'utilisateur doit choisir la suite à donner à l'action

« Apps are created equal »

Ce slogan mis en avant par Google signifie que les applications natives incluses de base sur le terminal et celles développées et installées par la suite sont sur le même plan, c'est-à-dire qu'elles ont les mêmes droits.

Le mécanisme de résolution des Intents en est une bonne preuve.

La résolution implicite est une fonctionnalité très puissante. Cependant, il est tout à fait légitime de vouloir appeler une activité précise et avoir l'assurance qu'aucune autre activité ne pourra intervenir à la place. Ceci arrive d'autant plus fréquemment que les activités à enchaîner se trouvent au sein de la même application.

Pour cela, la construction de l'objet *Intent* doit se faire en nommant l'activité au travers des méthodes `setComponent`, `setClassName` ou `setClass` :

```
Intent explicit = new Intent();
explicit.setClassName("org.florentgarin.android.intenttester",
    "org.florentgarin.android.intenttester.FakeActivity");
startActivity(explicit);
```

5.3.3 La communication entre activités

Comme on l'a vu lors des chapitres précédents, une activité A lance une activité B en lui transmettant un *Intent* qui est une sorte de message contenant toutes les informations nécessaires à B pour une bonne compréhension de ce que A attend de lui. En sens inverse, lorsque B aura achevé son traitement, il aura sans doute des informations à fournir à A pour rendre compte des opérations effectuées, en tout cas au minimum pour annoncer si le traitement s'est soldé par un succès ou un échec.

Si l'activité A a démarré l'activité B par la méthode *startActivity(Intent)*, la communication retour ne sera pas possible. A ne sera même pas alertée de la fin de l'activité B. Par recevoir cette notification, accompagnée éventuellement de données complémentaires, l'activité devra être amorcée par la méthode *startActivityForResult(Intent, int)*. L'entier passé en plus de l'*Intent* est un simple code qui sera renvoyé tel quel à l'activité A lors du retour, lui permettant d'identifier l'activité venant de s'achever.

La valeur de l'entier peut être librement choisie par l'activité émettant l'*Intent*, ce code est indispensable lorsque cette activité en lance plusieurs autres car les notifications de fin sont toutes acheminées par le même canal, par la méthode *onActivityResult*, il est donc crucial de pouvoir distinguer la provenance du message de retour.

La figure 5.3 montre un scénario complet d'une collaboration entre deux activités. Le point de départ est le lancement de l'*Intent* et le point d'arrivée est la réception de la notification de retour par la première activité. Les flèches en pointillés reliant les deux activités indiquent clairement que le couplage entre les activités est faible : aucune des deux ne connaît l'autre, elles communiquent par messages interposés acheminés par le framework d'Android.

Au niveau du code source, l'étape (1) serait (dans la classe de l'activité A) :

```
Intent intent = new Intent();
intent.setClassName("org.florentgarin.android.intenttester",
    "org.florentgarin.android.intenttester.FakeActivity");
startActivityForResult(intent, FAKE_REQUEST_CODE);
```

Avec la définition de la constante suivante (qui ne requiert pas de visibilité de type public) :

```
private static final int FAKE_REQUEST_CODE = 0;
```

L'étape (2) est prise en charge par Android et non implémentée par du code applicatif. Ensuite, en (3) et (4), dans la méthode *onCreate* de l'activité B, on aurait :

```
Intent intent = getIntent();
//...
//intent.getData();
//traitement...
//intent.setData(data);
//...
setResult(RESULT_OK, intent);
finish();
```

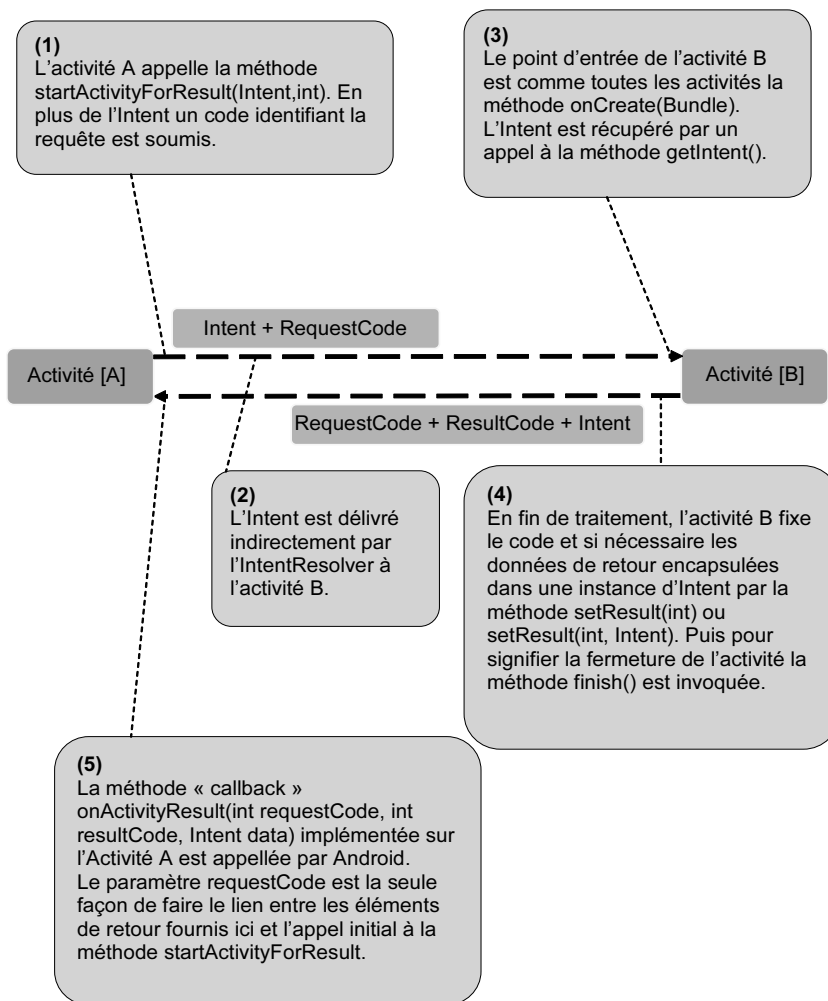


Figure 5.3 — Communication inter-activités

Enfin, la méthode `onActivityResult` de A s'exécute avec les informations de résultat en paramètre :

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data){
    if (requestCode == FAKE_REQUEST_CODE) {
        final TextView resultText = (TextView) findViewById(R.id.resultText);
        //...
        //traitement du retour
        //data.getData();
        //...
        resultText.setText(RESULT_OK==resultCode?"Succès":"Echec");
    }
}
```

5.4 LES SERVICES

À l'instar des activités, les services sont des composants qui peuvent être démarrés depuis l'application à laquelle ils appartiennent ou à distance depuis un programme tiers. Les ressemblances s'arrêtent là, les services n'ont pas de représentation visuelle et la logique qui les gouverne diffère.

Le rôle d'un service est de piloter une tâche de fond dont l'exécution est décorrélée de toute activité bien qu'une ou plusieurs activités puissent diriger ce service. L'exemple classique est la lecture de morceaux de musique par la fonction balladeur du téléphone. Une activité permet de créer sa liste de lecture, de la mettre en pause ou de naviguer entre les entrées de celle-ci. Par contre, il est bien appréciable que la musique continue même après que l'on ait quitté la lecture multimédia et que l'on soit en train de naviguer sur Internet ou de parcourir sa boîte mail.

Dans le principe, les services Android ressemblent à n'importe quel autre système RPC comme COM, Corba ou RMI. L'idée est de définir un service, c'est-à-dire un objet exposant des méthodes sollicitées à distance depuis un autre processus. Les méthodes sont appelées exactement de la même façon que les autres objets dont le code est exécuté localement.

Les méthodes « remote » peuvent accepter des objets en paramètres et peuvent également retourner d'autres objets et même lancer des exceptions en cas d'erreur comme n'importe quelle méthode. Dans le cas de méthodes de service distantes, toutes ces interactions donnent lieu à des échanges interprocessus, les objets passent d'un processus à l'autre par des opérations de marshalling et unmarshalling. Toutefois, cette complexité est masquée au client du service qui, comme on l'a dit, se contente de dialoguer avec le service en appelant simplement ses méthodes. Côté serveur aussi, les difficultés d'implémentation du protocole sont prises en charge par le framework.

5.4.1 Le langage AIDL

Pour définir un service, la première étape est de définir l'interface de celui-ci : les méthodes distantes et les objets transitant entre le client et le serveur. La définition de cette interface ne se fait pas en Java mais dans un langage spécifiquement conçu à cet effet, l'IDL (*Android Interface Definition Language*).

L'IDL est différent de l'IDL de Corba. En fait, l'IDL est plus simple à appréhender dans la mesure où la syntaxe de ce langage est très proche de celle de Java.

Par rapport à une interface Java standard, il y a quelques contraintes qui pèsent sur la définition des interfaces IDL. En outre, trois mots clés : « in », « out » et « inout » ont été introduits. Ces mots clés se positionnent devant les paramètres des méthodes et pour indiquer leur direction.

La signification de ces mots clés est la suivante :

- « in » marque les paramètres dont la valeur est passée du client vers le serveur ;
- « out » ceux dont la valeur est fixée par le serveur et retournée au client ;
- « inout » enfin concerne les paramètres initialement transportés du client vers le serveur mais qui peuvent être modifiés par celui-ci ; ces modifications seront répercutées côté client.

En dehors du concept du sens des paramètres complètement étranger à Java, les interfaces AIDL se doivent de respecter certaines règles quant au choix des types des paramètres et des objets renvoyés.

Seuls les types suivants sont autorisés :

- **Les types primitifs Java (boolean, byte, char, int, long, float, double) à l'exception de short** – Ces types ne peuvent être que des paramètres « in » et non des « out » ni des « inout ». L'ajout du modificateur « in » est donc optionnel. Si le service souhaite transmettre en résultat de la méthode un type primitif Java, il faudra le définir en retour de la méthode.
- **Le type String** – En AIDL, ce type est considéré comme un primitif : il n'apparaît qu'en paramètre « in ».
- **Le type CharSequence** – Comme le String, il ne peut s'agir que d'un paramètre « in ».
- **Le type List** – Bien qu'en Java l'interface *List* soit définie dans le package *java.util*, il n'est pas nécessaire d'importer cette interface dans le fichier AIDL. Il est possible d'utiliser la forme générique du type et de préciser la nature des objets à contenir. Évidemment, pour pouvoir être échangée entre le client et le serveur, la liste ne pourra regrouper que des objets appartenant individuellement aux types autorisés (primitifs, String, *CharSequence*...). Les listes peuvent être employées comme paramètre « in », « out » et « inout ».
- **Le type Map** – Cette structure de données qui, on le rappelle, permet d'indexer des valeurs par des clés, fonctionne en AIDL comme la *List*, à ce détail près que la forme générique n'est pas autorisée.
- N'importe quel **objet Parcelable**.
- Toutes les autres interfaces générées à partir d'une **définition AIDL**.

Ces deux derniers items méritent une explication approfondie.

Les objets Parcelable

Parcelable est une interface qui, à la manière de l'interface *Serializable*, caractérise les objets ayant la capacité à transiter entre plusieurs processus. Pour que le marshalling/unmarshalling de ces objets fonctionne en plus d'implémenter les méthodes de l'interface *Parcelable*, il faudra définir dans ces objets un attribut static *CREATOR* implémentant *Parcelable.Creator*. Cet attribut sera la factory de l'objet, une interface ne pouvant pas contraindre à la déclaration d'une méthode statique ; c'est l'astuce de conception employée par les équipes Android.

Un exemple valant mieux que de longs discours, voici un objet Invoice (facture) implémentant *Parcelable* :

```
package org.florentgarin.android.service;
import android.os.Parcel;
import android.os.Parcelable;
public final class Invoice implements Parcelable {
    private int number;
    private int vendorCode;
    private int totalAmount;
    private String comment;
    public static final Parcelable.Creator<Invoice> CREATOR = new
Parcelable.Creator<Invoice>() {
        @Override
        public Invoice createFromParcel(Parcel in) {
            Invoice invoice = new Invoice();
            invoice.number = in.readInt();
            invoice.vendorCode = in.readInt();
            invoice.totalAmount = in.readInt();
            invoice.comment = in.readString();
            return invoice;
        }
        @Override
        public Invoice[] newArray(int size) {
            return new Invoice[size];
        }
    };
    public Invoice() {
    }
    @Override
    public int describeContents() {
        return 0;
    }
    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeInt(number);
        dest.writeInt(vendorCode);
        dest.writeInt(totalAmount);
        dest.writeString(comment);
    }
}
```

À la base, l'objet Invoice est un banal Bean modélisant un concept métier, en l'occurrence une facture. Cet objet a quatre attributs : number, vendorCode, totalAmount et comment. Les getters et setters ne sont pas ici présents, uniquement pour ne pas alourdir le listing de code, mais dans la réalité il en faudra bien, à moins de déclarer ces attributs publics ce qui est plus performant, bien que cassant l'encapsulation objet.

Les méthodes de *Parcelable* à implémenter sont au nombre de deux :

- **int describeContents()** - Cette méthode dont l'explication fournie dans la javadoc est quelque peu succincte se doit de retourner un masque de bits informant d'éventuels types particuliers manipulés par cet objet *Parcelable*. Dans la pratique, cette méthode sert rarement ; généralement, on se contente donc de retourner 0.

- **writeToParcel(Parcel dest, int flags)** - Le rôle de cette méthode est d'écrire l'état de l'objet dans ce qu'Android nomme une parcelle. Un objet *Parcel* est un conteneur qui peut être échangé avec un objet de type *IBinder* qui est l'interface de base des objets invocables à distance comme les services.
- L'implémentation de cette méthode consiste donc à écrire un à un sur la *Parcel* passée en paramètre les attributs de l'objet. La parcelle offre pour cela un ensemble de méthodes *writeXXX* pour chacun des types des attributs, un peu sous le même modèle que l'*ObjectOutputStream* de Java.

Ensuite, la parcelle sera transmise à l'autre bout du canal IPC pour que l'objet puisse être recréé. Le deuxième paramètre de la méthode est un flag de type *int*. Si la valeur de ce flag est égale à la constante *Parcelable.PARCELABLE_WRITE_RETURN_VALUE*, cela signifie que l'opération courante de sérialisation prend part à la construction d'une valeur de retour d'une méthode. L'objet sauvegardé sur la parcelle est donc soit retourné directement par une méthode remote soit par un paramètre « out » ou « inout ». Dans l'exemple de la facture, et assez fréquemment d'ailleurs, ce flag peut être ignoré. Mais on pourrait toutefois imaginer que la sérialisation d'un objet lorsqu'il intervient en tant que valeur de retour donne lieu à un traitement différent que lorsqu'il apparaît comme paramètre « in ».

Une fois l'interface *Parcelable* implémentée par l'objet, il faut créer la « factory » sous forme d'un attribut static nommé *CREATOR* qui implémentera l'interface *Parcelable.Creator<T>*, *T* étant le type générique, la classe *Invoice* dans l'exemple.

Cela ne pose pas de difficultés particulières, l'interface ne possédant que deux méthodes :

- **T[] newArray(int size)** - Cette méthode renvoie un tableau typé vide (les entrées valent toutes null) de la taille fixée en paramètre.
- **T createFromParcel(Parcel in)** - Cette méthode est le miroir de la méthode *writeToParcel*. Son objectif est d'instancier l'objet *Parcelable* et de lire la valeur de ses attributs depuis la parcelle. Il est capital que la lecture se fasse exactement dans le même ordre que l'écriture.

Voilà, l'objet *Invoice* est dorénavant apte à pouvoir être échangé au travers d'un service. Pour pouvoir l'employer dans la définition AIDL, il ne reste plus qu'à générer un fichier un peu spécial : le fichier « *project.aidl* ». Ce fichier référence les objets *Parcelable* de sorte à pouvoir les utiliser dans les définitions des méthodes des services. Pour cela, il faut faire un clic droit sur le projet puis : *Android Tools > Create Aidl preprocess file for Parcelable classes* (figure 5.4).

Le fichier *project.aidl* créé est le suivant :

```
// This file is auto-generated by the
// 'Create Aidl preprocess file for Parcelable classes'
// action. Do not modify!
parcelable org.florentgarin.android.service.Invoice
```

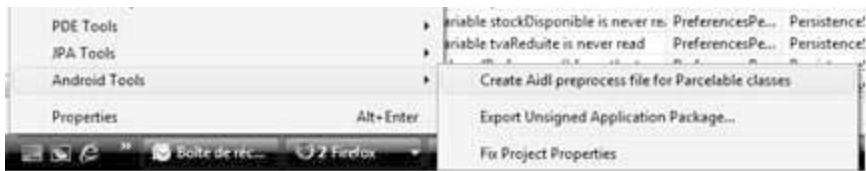


Figure 5.4 — Création du fichier AIDL

Alternativement, si l'on n'utilise pas eclipse, on peut créer soit même un fichier AIDL par type *Parcelable*. Par exemple le fichier Invoice.aidl serait :

```
package org.florentgarin.android.service;
parcelable Invoice;
```

Les types *IBinder*

Android permet de véhiculer au travers d'interface de service, en plus des types déjà cités, des objets « remote ». C'est un peu comme si on passait en paramètre d'un service un autre service. Cette fonctionnalité est assez répandue dans les systèmes RPC, c'est quelque chose qu'il est possible de faire par exemple en Java RMI où un *UnicastRemoteObject* peut être échangé sur le réseau.

Cependant, il y a une différence fondamentale entre le marshalling d'une primitive ou d'un *Parcelable* et celui d'un *IBinder*. Dans le premier cas, la valeur est transmise par copie et dans le second cas, par référence. Ainsi lorsqu'un objet remote est passé en paramètre d'un service, l'objet réellement réceptionné à l'autre bout du tuyau est un proxy. Concrètement, cela veut dire que les méthodes qui sont sur le type *IBinder* sont elles-mêmes distantes et leurs invocations engendreront à nouveau une communication IPC dans l'autre sens que celui dans lequel l'objet a été envoyé !

Définition du service

Maintenant que tous les objets prenant part aux échanges entre les clients et le service ont été construits, on est fin prêt à écrire le fichier d'interface AIDL du service :

```
package org.florentgarin.android.service;
interface ISampleService {
    String echo(in String message);
    int max(int a, int b);
    void duplicate(int index, inout long[] data);
    Invoice getInvoice(int number);
}
```

Ce fichier doit porter le nom du service, *ISampleService.aidl* et doit être placé dans le répertoire correspondant à son package.

La présence de ce fichier est détectée automatiquement par le *plug-in* Eclipse qui génère alors les Stub et Proxy, contenus dans le fichier source *ISampleService.java*, qui serviront de point de départ à l'implémentation du service. Si l'application est développée avec un autre IDE qu'Eclipse, il est obligatoire d'exécuter la commande *aidl* pour générer ce fichier.

Vocabulaire RPC

Les termes employés dans ce chapitre sont souvent empruntés au vocabulaire des systèmes objets distribués comme Corba, RMI, SOAP...

Par exemple, le **marshalling** est l'action visant à transformer une représentation mémoire d'un objet ou d'un graphe d'objets dans un format permettant la transmission de cet objet à un autre programme.

L'opération de **unmarshalling** est l'opposé du marshalling. Il s'agit de recréer en mémoire l'objet initial à partir du format de transmission.

Les mots stub et proxy apparaissent aussi dans ce chapitre. La distinction entre ces deux termes n'est pas souvent très claire dans la littérature informatique où ils sont parfois employés indifféremment pour représenter la même notion. Le proxy, mandataire en français, est un objet agissant à la place d'un autre. Dans le contexte d'un système d'objets distribués tel que celui des services Android, il s'agit de l'objet qui sera instancié côté client, qui offrira les mêmes méthodes que celles de l'interface du service. Cet objet sera chargé de gérer la communication interprocessus, marshalling et unmarshalling des objets, avec le service. Il donnera ainsi l'impression au consommateur du service que celui-ci se comporte comme un simple objet instancié localement.

Le stub (souche) est la classe embryon qui sera étendue pour fournir l'implémentation complète du service. Cette classe propose également quelques méthodes statiques notamment « asInterface » utilisée sur le client pour obtenir le service (en réalité le proxy !) sous la forme de son interface.

5.4.2 Implémentation du service

L'implémentation du service se fait en étendant la classe interne Stub générée. Dans l'exemple, il s'agit de `ISampleService.Stub` et le nom choisi pour cette implémentation est `SampleServiceImpl` dont voici la source :

```
package org.florentgarin.android.service.impl;
import org.florentgarin.android.service.ISampleService.Stub;
import android.os.RemoteException;
public class SampleServiceImpl extends Stub {
    @Override
    public void duplicate(int index, long[] data) throws RemoteException {
        long valueToCopy=data[index];
        for(int i=0;i<data.length;i++)
            data[i]=valueToCopy;
        //data est un paramètre inout. Les modifications apportées
        //se verront sur le client.
    }
    @Override
    public String echo(String message) throws RemoteException {
        return message;
    }
    @Override
    public Invoice getInvoice(int number) throws RemoteException {
        Invoice invoice = new Invoice();
        invoice.setNumber(number);
        invoice.setComment("impayée !");
    }
}
```

```

        invoice.setTotalAmount(15000);
        invoice.setVendorCode(12345);
        return invoice;
        //dans une véritable application on effectuerait probablement
        //une recherche dans un content provider ou une base de données
        //sur un serveur distant.
    }
    @Override
    public int max(int a, int b) throws RemoteException {
        return Math.max(a,b);
    }
}

```

La tâche d'implémentation semble relativement triviale, néanmoins on peut émettre quelques recommandations. Premièrement, le service devra être *thread safe*. En effet, à la manière d'un serveur d'application, Android gère un pool de *threads* chargés de l'exécution, à la demande des clients, des méthodes « remote ». Par contre, contrairement justement aux serveurs d'applications Java EE, il n'y a pas ici d'équivalent au pool d'EJB, autrement dit plusieurs threads pourront potentiellement exécuter des méthodes en même temps sur la même instance du service.

En outre, l'autre conséquence de ces traitements en parallèle sur la même instance d'objet est que le service ne peut gérer d'état.

5.4.3 Publication du service

Le service est défini et implémenté mais il n'est pour l'instant atteignable par aucun client. En réalité, le service n'est pour l'instant qu'un *IBinder* et pas encore un vrai service. L'ultime étape est d'autoriser un client à en récupérer une instance. Dans le monde Corba, les services doivent s'enregistrer auprès d'un annuaire. Ainsi, lorsqu'un client interroge cet annuaire en passant un nom, il récupère une instance du service correspondant, ou plutôt son proxy, et peut ensuite l'interroger.

Dans Android, c'est un peu la même histoire. Le service sera accessible non pas en fournissant un nom mais un objet *Intent*. En outre, le service peut déclarer plusieurs *IntentFilters* ; de ce fait il pourra être sollicité par des *Intents* différents. Néanmoins, avant de paramétrer les *IntentFilters*, la publication exige d'implémenter à nouveau une interface, il s'agit de « *android.app.Service* » :

```

package org.florentgarin.android.service.impl;
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
public class SampleService extends Service {
    private final SampleServiceImpl service=new SampleServiceImpl();
    @Override
    public IBinder onBind(Intent intent) {
        return service;
    }
}

```

Il n'y a qu'une méthode à implémenter : la méthode *onBind* qui renvoie l'implémentation du service en fonction de l'*Intent*. Cela peut sembler étrange, on pourrait se demander à quoi peut bien servir cette méthode et l'intérêt de passer l'*Intent*. En fait la réponse tient dans le fait qu'un service Android peut regrouper plusieurs objets « remote ». Pour faire une comparaison, c'est un peu comme les web services SOAP qui peuvent définir plusieurs Ports.

Dans l'exemple, le service n'encapsule qu'une seule interface AIDL, l'implémentation est donc des plus simples et il n'est même pas utile d'examiner l'*Intent*.

Après cela il faut déclarer le service dans le manifeste :

```
<service android:name=".impl.SampleService"
        android:exported="true">
    <intent-filter>
        <action
            android:name="org.florentgarin.android.service.ISampleService" />
        </intent-filter>
    </service>
```

Le filter est important, sans cela le service ne pourra pas être appelé depuis une autre application mais uniquement en interne.

Juste avant de passer à l'implémentation du client, il faut penser à packager un jar contenant les classes dont celui-ci aura besoin. Il s'agit uniquement, dans l'exemple, des classes Invoice et ISampleService. Les classes du package *org.florentgarin.android.service.impl* ne sont pas à mettre dans le jar car ces classes (SampleService et SampleServiceImpl) sont utilisées côté service exclusivement.

5.4.4 Côté client

Le projet Android consommateur du service ne devra pas manquer d'importer en tant que librairie le fameux jar contenant le Stub et les classes de type *Parcelable*.

Pour obtenir une référence vers le service, il faut utiliser la méthode *bindService(Intent, ServiceConnection, int)*.

Le premier paramètre est l'*Intent* dont l'action doit correspondre à celle de l'*IntentFilter*, comme pour les *Activity*. Dans l'exemple, c'est le nom qualifié de l'interface du service *ISampleService*.

Le deuxième paramètre est une implémentation de l'interface *ServiceConnection*. Cette classe sera le callback de l'opération de binding. Ainsi la méthode *onServiceConnected(ComponentName, IBinder)* sera invoquée par le système à la connexion et la méthode *onServiceDisconnected* à la déconnexion. Le rôle premier de cette classe est de capter le service, fourni sous forme d'*IBinder*. Ensuite, il reste à appeler la méthode *ISampleService.Stub.asInterface(binder)* qui se chargera de caster le binder dans le type de l'interface du service.

Le troisième et dernier paramètre est un flag d'option sur le traitement. La valeur *Context.BIND_AUTO_CREATE* employée indique que le service devra être lancé si ce n'est pas le cas.

Au final, un exemple complet d'activité cliente du service pourrait être :

```
package org.florentgarin.android.service.client;
import org.florentgarin.android.service.ISampleService;
import android.app.Activity;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.IBinder;
import android.os.RemoteException;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
public class ServiceClientActivity extends Activity {
    private ISampleService service;
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        ServiceConnection connection = new ServiceConnection() {
            @Override
            public void onServiceConnected(ComponentName className, IBinder
binder) {
                service = ISampleService.Stub.asInterface(binder);
            }
            @Override
            public void onServiceDisconnected(ComponentName className) {
                service=null;
            }
        };
        bindService(new Intent(ISampleService.class.getName()),
            connection, Context.BIND_AUTO_CREATE);
        final Button sendBtn = (Button) findViewById(R.id.sendCmdBtn);
        final TextView result = (TextView) findViewById(R.id.result);
        sendBtn.setOnClickListener(new View.OnClickListener(){
            @Override
            public void onClick(View v) {
                try {
                    result.setText("le plus grand entre 5 et 10
est : " + service.max(5,10));
                } catch (RemoteException e) {
                    e.printStackTrace();
                }
            }
        });
    }
}
```

L'activité comporte un bouton et une zone de texte. Lorsque l'utilisateur clique sur le bouton, la méthode du service `max` qui compare deux entiers est appelée. La valeur retournée par le service est ensuite affichée dans la zone de texte.

Avant de lancer le client, il faut bien entendu avoir déployé l'application contenant le service. Le résultat obtenu est alors le suivant :

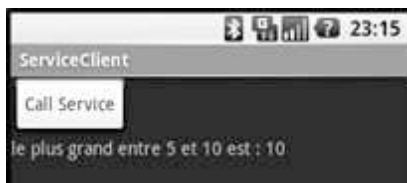


Figure 5.5 — Le résultat du service est affiché par l'activité

5.5 LE BUS DE MESSAGES

Android, dont l'une des forces est d'offrir un environnement propice à la collaboration entre les applications, ne pouvait pas ne pas être doté d'un bus de messages.

La pièce centrale de ce bus de messages est le *BroadcastReceiver* qui, avec les classes *Activity*, *Service* et *ContentProvider* (étudié au chapitre consacré à la persistance), fait partie de la famille des composants Android.

Les objets de type *BroadcastReceiver*, comme les activités et les services, réagissent aux intentions. Cependant, leur fonctionnement se caractérise par le fait qu'aucun comportement particulier n'est attendu du *BroadcastReceiver* par l'émetteur de l'*Intent*. Une autre particularité, toujours en comparaison des activités et des services, est qu'un même *Intent* pourrait être reçu par plusieurs *BroadcastReceivers*. L'*Intent* s'apparente ici à un événement diffusé sur l'ensemble du système Android ; libre à chacun de l'ignorer ou de le capter pour en faire ce que bon lui semble.

5.5.1 Deux modes d'émission

Un mode classique

Il existe deux manières d'émettre un *Intent* en direction des *BroadcastReceiver*. Il y a l'envoi classique où les récepteurs qui se sont abonnés au type d'*Intent* sont notifiés dans un ordre indéterminé, souvent simultanément. Assurément en tout cas, ces receivers réagissent en étant isolés hermétiquement les uns des autres. Pour diffuser un événement quelques lignes suffisent :

```
Intent callEvent=new Intent("android.intent.action.PHONE_STATE");  
sendBroadcast(callEvent);
```

La méthode `sendBroadcast` est disponible sur la classe `Context`.

Un mode ordonné

Le second mode de diffusion s'éloigne légèrement de l'idée que l'on pourrait se faire du broadcast. En effet, dans ce mode, les receivers sont informés de l'arrivée de l'*Intent* les uns après les autres en fonction d'une priorité que les receivers se sont eux-mêmes attribués par l'attribut « android:priority » fixé dans le manifeste sur l'*IntentFilter* qui leur a permis d'être informés de l'événement.

De plus, dans ce mode, le *BroadcastReceiver* qui est en train de traiter l'*Intent* a la possibilité d'annuler le broadcast et de casser ainsi la chaîne de notification grâce à l'appel à la méthode *abortBroadcast*. Il peut aussi positionner un code de type entier, méthode *setResultCode(int)*, ou des données qui seront lues par le prochain *BroadcastReceiver*, *setResultData(String)* et *setResultExtras(Bundle)*, et tout logiquement il peut lire ces mêmes informations renseignées par le composant précédent (méthodes *get* des *setters* précédemment cités). Les *BroadcastReceivers* communiquent donc entre eux mais sans se connaître et sans savoir clairement leur propre position dans la chaîne.

L'ultime *BroadcastReceiver* pourra, si nécessaire et de manière garantie, être fourni par l'émetteur. Généralement, son rôle est de recueillir le code et les données dans leur état final après avoir été manipulés par toute la chaîne des *BroadcastReceivers*.

L'envoi ordonné d'un *Intent* se fait en appelant la méthode *sendOrderedBroadcast*. C'est dans cette méthode qu'il faudra passer en paramètre le dernier *BroadcastReceiver*.

Quel que soit le mode d'émission retenu, classique ou ordonné, il existe pour les deux méthodes *send* des versions surchargées ; celles-ci offrent l'opportunité de préciser la permission qui devra avoir été accordée aux *BroadcastReceivers* pour qu'ils puissent recevoir l'*Intent*.

Le mode « ordonné » d'émission d'*Intent* peut réserver quelques surprises. En effet, l'ordre dans lequel les récepteurs sont informés n'est pas maîtrisable, il suffit par exemple que sur un système Android donné, une application quelconque soit déployée avec un *BroadcastReceiver* déclarant une certaine priorité pour que cet ordre se retrouve chamboulé. Il est donc primordial de ne pas présumer de cet ordre en implémentant un algorithme qui en dépendrait.

5.5.2 Deux façons de s'abonner aux événements

Pour être en mesure de recevoir des intentions, les *BroadcastReceivers* doivent s'enregistrer sur le système en indiquant les *Intents* qu'ils souhaitent écouter.

Pour ce faire, on peut ajouter un élément « receiver » dans le manifeste en précisant la liste de ses *IntentFilters*, chacun d'entre eux ayant un nom, et potentiellement une catégorie et une entrée « data » comme c'était déjà le cas pour les activités ou les services. Une fois l'application déployée, ces *BroadcastReceivers* sont constamment en mesure de recevoir les événements les intéressants.

L'exemple ci-dessous enregistre le receiver implémenté par la classe `SMSBroadcastReceiver`. Un filtre a été défini afin de recevoir les Intents « `android.provider.Telephony.SMS_RECEIVED` » :

```
<receiver android:name=".SMSBroadcastReceiver">
    <intent-filter>
        <action android:name="android.provider.Telephony.SMS_RECEIVED" />
    </intent-filter>
</receiver>
```

Si toutes les applications peuvent lancer leurs propres événements, il en existe déjà un certain nombre qui sont nativement diffusés par le système Android. Celui-ci en fait partie. Il sert à notifier de l'arrivée d'un SMS.

En plus de la publication statique du `BroadcastReceiver`, il est possible de réaliser cet enregistrement dynamiquement par du code grâce à la méthode de l'objet `Context` `registerReceiver`. La méthode `unregisterReceiver`, comme son nom le laisse présager, désenregistre le receiver.

Grâce à ces méthodes, il est possible de maîtriser les périodes pendant lesquelles les événements peuvent être reçus. Cette fonctionnalité est particulièrement intéressante lorsque l'on souhaite lier la réception des Intents au cycle de vie d'une activité. Par exemple, si une application nécessite pour son fonctionnement de capter des événements qui seront affichés à l'écran, il n'est pas utile de maintenir l'écoute pendant les phases où cette activité n'est pas visible. Il ne faut oublier qu'un receiver référencé par le manifeste de son application est en permanence en capacité d'être sollicité par les Intents, même si le processus de son application n'est pas lancé ; le système s'en chargerait si c'était le cas. Donc, une stratégie pertinente pour un receiver fonctionnellement couplé à une activité serait de l'enregistrer dans la méthode `onResume` de l'activité et de le désenregistrer dans sa méthode `onPause`.

Pour revenir à l'exemple, si l'ajout de l'`IntentFilter` dans le manifeste fait du receiver un candidat aux notifications de réception de SMS, cela ne suffit pas entièrement.

Pour être opérationnel et effectivement recevoir l'`Intent`, une permission est exigée, le manifeste doit donc être complété de la façon suivante :

```
<uses-permission
    android:name="android.permission.RECEIVE_SMS"></uses-permission>
```

Cette permission octroie des droits au `BroadcastReceiver` pour satisfaire l'émetteur. Réciproquement, le `BroadcastReceiver` peut aussi exiger de l'émetteur certaines permissions pour qu'il puisse le solliciter. Cela se fait encore par le biais d'une déclaration dans le manifeste ou par la méthode `registerReceiver` pour les receivers dynamiquement enregistrés.

5.5.3 Implémentation du receiver

Tout receiver doit étendre la classe abstraite *BroadcastReceiver*.

Cette classe possède plusieurs méthodes mais une seule est abstraite et requiert d'être implémentée, c'est la méthode *onReceive(Context, Intent)*.

Cette méthode contient le code qui sera exécuté quand un *Intent* compatible avec un *IntentFilter* du receiver aura été émis. Cette méthode est l'équivalente de la méthode *onMessage* de l'interface *MessageListener* de l'API JMS (Java Message Service) pour ceux qui connaissent.

Voici une implémentation basique qui affiche à l'écran le SMS reçu ainsi que le numéro de téléphone de l'expéditeur :

```
package org.florentgarin.android.broadcastreceiver;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.telephony.gsm.SmsMessage;
import android.widget.Toast;
public class SMSBroadcastReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Bundle extras = intent.getExtras();
        SmsMessage[] sms = null;
        if (extras != null){
            Object[] pdus = (Object[]) extras.get("pdus");
            sms = new SmsMessage[pdus.length];
            for (int i=0; i<pdus.length; i++){
                sms[i] = SmsMessage.createFromPdu((byte[])pdus[i]);
            }
            String message = "SMS reçu de : ";
            message += sms[0].getOriginatingAddress();
            message += "\n";
            message += sms[0].getMessageBody();
            Toast.makeText(context, message, Toast.LENGTH_LONG).show();
        }
    }
}
```

L'*Intent* contient un objet de type *Bundle* qui encapsule les données du SMS. L'objet *SmsMessage* qui appartient à l'API de téléphonie sera analysé plus en profondeur au paragraphe « Les fonctions de téléphonie ».

Ensuite, les informations sont restituées à l'utilisateur grâce à la classe *Toast* dont le rôle est de montrer à l'écran de rapides messages.

Pour tester ce programme d'exemple, il faut non seulement déployer l'application sur Android mais aussi générer l'événement *android.provider.Telephony.SMS_RECEIVED*.

Pour diffuser l'événement, il y a deux possibilités : soit recourir à la méthode *sendBroadcast(Intent)*, soit utiliser une commande de l'émulateur pour simuler l'envoi de SMS.

Cette dernière option paraît plus probante pour réaliser la démonstration de la réception d'un SMS car elle part véritablement de l'action.

Une fois l'émulateur lancé et l'application du *BroadcastReceiver* installée, il faut ouvrir une session Telnet :

```
telnet localhost 5554
```

Puis saisir la commande d'envoi de SMS :

```
sms send 0612345678 Salut, le RDV tient toujours ?
```

Pour les utilisateurs d'Eclipse, cette commande peut être émise directement depuis l'IDE à partir de la perspective DDMS (Dalvik Debug Monitor Service) :

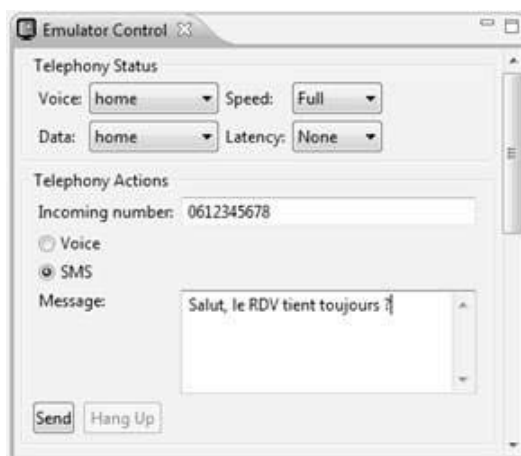


Figure 5.6 — La vue "Emulator Control" du *Plug-in* Eclipse

Le numéro de téléphone à saisir n'est pas celui à qui le message est destiné mais celui avec lequel le SMS sera supposé avoir été envoyé. Le destinataire du SMS est l'émulateur auquel le DDMS est connecté, il ne s'agit bien que d'une fonction de simulation de réception de messages et en aucun cas d'un outil pour envoyer de vrais SMS.

Bien sûr, si l'application tourne sur un vrai terminal, on a aussi la possibilité d'envoyer un vrai SMS. Le résultat produit sera le même (figure 5.7).

Chose intéressante, le SMS apparaît à deux endroits différents : dans la barre de notification et sous forme de popup.

L'affichage dans la barre de notification est l'œuvre d'Android et le popup est affiché par la classe *Toast* du *SMSBroadcastReceiver* qu'on vient de déployer. Ceci démontre effectivement que plusieurs receivers peuvent écouter le même événement.

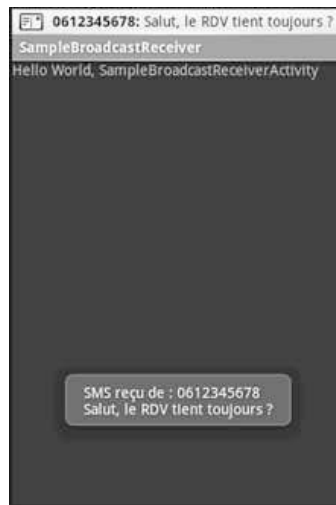


Figure 5.7 — Réception du SMS

5.5.4 Une durée de vie très courte

La méthode *onReceive* du *BroadcastReceiver* est exécutée par le *thread main* du processus du receiver. Ce thread est également celui chargé de l’affichage des activités hébergées par le processus. La règle est donc toujours la même : il faut éviter au maximum de monopoliser ce thread, seules les tâches de courtes durées doivent y être lancées.

En plus de ce principe général, une contrainte supplémentaire pèse sur les *BroadcastReceivers*, leur durée d’exécution ne doit pas excéder 10 secondes faute de quoi une boîte de dialogue ANR (*Application Not Responding*) apparaîtra.

Par exemple l’ajout de ces lignes dans la méthode *onReceive* :

```
try {  
    Thread.sleep(15000);  
} catch (InterruptedException e) {}
```

Afin de simuler un traitement long, il génère l’affichage de l’écran figure 5.8.

Ceci a pour objectif de protéger le téléphone des applications mal conçues qui pourraient perturber l’ensemble du système.

Face à une boîte de dialogue ANR, l’utilisateur a toujours la possibilité d’accorder du temps supplémentaire au traitement en cliquant sur le bouton « Wait » (attendre). Toutefois cette intervention sera à n’en pas douter très mal perçue par l’utilisateur qui aura le sentiment que l’application est « plantée ». La limite des 10 secondes ne doit donc jamais être franchie.

Comment faire alors si on doit dérouler un long traitement suite à la réception d’un événement ?



Figure 5.8 — Fenêtre d'ANR

Le premier réflexe consiste simplement à créer un *thread* dévolu à cette tâche (on désigne couramment ce genre de thread sous l'intitulé « *worker thread* »). Malheureusement, ceci est une très mauvaise idée ! En effet, Android considère un *BroadcastReceiver* comme inactif à partir du moment où la méthode *onReceive* a retourné. Donc, si le processus propriétaire n'accueille aucun autre composant (ni activité ni service), il sera étiqueté « processus vide » et aura par conséquent de forte chance d'être tué par le système et cela même si le worker thread est toujours en activité.

On peut généraliser en disant qu'il est très hasardeux de lancer des opérations asynchrones dans la méthode *onReceive*. Par exemple, la méthode *bindService(Intent, ServiceConnection)* du contexte est formellement à éviter car celle-ci retourne immédiatement, avant même d'avoir pu joindre le service, l'objet de type *ServiceConnection* étant le callback appelé lorsque la connexion est effective.

La paire gagnante : *BroadcastReceiver* + *Service*

Pour répondre à ces deux impératifs qui peuvent sembler antinomiques, la non-monopolisation du thread principal et le maintien du processus, la solution à préconiser est de déléguer le travail à un service local qui sera sollicité par la méthode synchrone *startService(Intent)* du contexte.

Ainsi, l'existence de ce service prémunira le processus d'une morte subite ! Attention toutefois, la méthode *startService* déclenchera la méthode *onStart* sur le service. Cette méthode sera elle-même exécutée par le thread principal, ce qui comme on l'a déjà dit n'est pas souhaitable. Il est donc toujours nécessaire d'utiliser un worker thread, la différence résidant dans le fait que le traitement étant piloté dorénavant par le service, il n'y a plus de risque de voir le processus sauvagement abrégé.

Ce qu'il faut retenir

Ce chapitre est essentiel à la bonne compréhension d'Android. Les quatre composants que sont *Activity*, *Service*, *BroadcastReceiver* et *ContentProvider* sont les piliers d'Android. Contrairement à une simple classe lambda, ces composants sont activables à distance, chacun à leur façon, selon des conditions définies dans le manifeste. Si le processus hôte du composant interrogé n'est pas démarré, Android se chargera de le lancer. Les applications Android sont donc à multiples facettes et à multiples points d'entrées : en un mot modulaires.

Grâce à cette modularité, l'implémentation d'application par un jeu de construction d'éléments existants est réellement possible. On n'est plus sans cesse en train de réimplémenter les mêmes choses ; les écrans, les données, les fonctions sont recyclées entre les programmes. En outre, l'intégration des composants est très aboutie, l'utilisateur n'ayant pas conscience que l'exécution du logiciel s'étale sur plusieurs processus.

La notion même d'application s'en retrouve quelque peu bousculée...

6

La persistance des données

Objectifs

Ce chapitre traite de la conservation de l'état des applications d'une exécution à l'autre. Dit autrement, le sujet du chapitre est le stockage longue durée des données du programme sur le terminal.

6.1 SAUVEGARDER L'ÉTAT DES APPLICATIONS

La persistance des données est une composante importante des applications. Selon la nature de ces dernières, la masse d'information qu'elles seront amenées à lire et écrire pourra certes varier mais rares sont les applications ne nécessitant aucunement de sauvegarder leurs données.

Android propose plusieurs mécanismes pour gérer la persistance, c'est-à-dire lire et écrire des informations dont la durée de vie est supérieure au temps d'exécution du programme. Ces techniques de persistance, qui sont au nombre de trois, répondent à des problématiques différentes, le choix de l'une ou de l'autre dépend donc du contexte d'utilisation. Elles ont néanmoins toutes en commun d'offrir par défaut des espaces de persistance privés aux applications. En conséquence, une application X ne pourra accéder aux données de l'application Y que si celle-ci a expressément prévu ce scénario en fournissant un objet de type *ContentProvider*. Il s'agit là une nouvelle fois, au travers de ce principe, de garantir une sécurité optimale pour

l'utilisateur ; les applications peuvent communiquer entre elles mais il n'est pas question qu'un programme malveillant puisse occasionner des dégâts aux autres applications.

6.2 SYSTÈME DE FICHIERS

L'API d'Android permet de lire et écrire des fichiers, qu'ils soient stockés directement sur le téléphone ou sur un support amovible tel qu'une carte SD.

Comme pour les autres mécanismes de persistance, l'espace de stockage sur le système de fichiers est en principe cloisonné entre les applications ; c'est d'ailleurs pour cette raison que le point de départ pour effectuer des manipulations sur les fichiers est à chercher sur l'objet *Context*. Néanmoins, il est possible d'outrepasser cette séparation et de créer des fichiers qui pourront être partagés entre plusieurs applications.

6.2.1 Lecture d'un fichier

La lecture des fichiers se fait par l'interface standard *java.io.FileInputStream* :

```
FileInputStream inputStream = openFileInput("mon_fichier");
```

Le fichier en question doit forcément exister sinon une *FileNotFoundException* est lancée. Ensuite, c'est du classique :

```
BufferedInputStream in = new BufferedInputStream(inputStream);
final int bufferSize = 1024*8;
byte[] data = new byte[bufferSize];
int length;
while ((length = in.read(data)) != -1){
    //les octets lus sont contenus dans data
}
in.close();
```

Comme les bonnes pratiques l'exigent, il est préférable d'encapsuler le *FileInputStream* dans un buffer (*BufferedInputStream*), ainsi on évite de solliciter à chaque lecture le support physique. Après, on monte, par lots de 8 Ko, le contenu du fichier dans le tableau *data*. Il conviendra d'y apporter le traitement spécifique adéquat.

Enfin, à l'issue du traitement, il faut fermer le flux.

On peut noter que la méthode *openFileInput* n'accepte pas les noms de fichiers arborescents. Cette gestion du système de fichiers fait penser un peu aux sauvegardes des parties de certains jeux vidéo : en somme, seul un nom identifie le fichier à ouvrir ; l'emplacement réel où se trouve le fichier est masqué à l'utilisateur.

Les objets *Context* (pour rappel *Activity* étend la classe *Context*) proposent une méthode listant les noms de tous les fichiers présents dans la zone de stockage de l'application :

```
String[] files = fileList();
```

6.2.2 Écrire dans un fichier

L'écriture dans la zone du « file system » des applications se passe de façon analogue mais inversée à la lecture, c'est-à-dire à l'aide de l'API « Stream » de Java.

Pour récupérer un *FileOutputStream*, il faut appeler la méthode *openFileOutput*, toujours sur le *Context*. Cependant, contrairement à la méthode *openFileInput*, en plus du nom du fichier, il faut passer en paramètre le mode d'accès choisi parmi la liste suivante :

- `MODE_APPEND`
- `MODE_PRIVATE`
- `MODE_WORLD_READABLE`
- `MODE_WORLD_WRITEABLE`

Ces modes dictent la manière dont l'écriture dans le flux se déroulera. `MODE_APPEND` indique que les écritures s'ajouteront à la suite du fichier sans écraser son contenu existant.

Le `MODE_PRIVATE` est le mode classique ; le fichier est créé dans une zone privée non accessible par d'autres applications et s'il existe déjà, il sera écrasé.

En réalité, dire que le fichier ne pourra être accédé que par l'application l'ayant écrit est un petit raccourci. En effet, il ne faut pas oublier qu'Android tourne sur un noyau Linux. Par conséquent, la gestion des droits est liée au compte Linux exécutant le programme. Si deux applications utilisent le même « user ID » (à configurer dans le fichier manifeste), elles partageront la même zone de stockage de fichiers.

`MODE_WORLD_READABLE` et `MODE_WORLD_WRITEABLE` signifient que le fichier sera créé avec les droits nécessaires pour respectivement autoriser sa lecture et son écriture par n'importe quelles autres applications et cela même si elles ont leur propre « user ID ». Attention cependant, le fichier ne sera pas pour autant créé dans une sorte de zone de stockage globale, il résidera toujours dans la partie du système de fichier rattachée à l'application l'ayant créé.

Pour accéder à ce fichier, il faudra donc obtenir une référence vers contexte de l'application :

```
Context appCtx=createPackageContext(applicationPackage,  
Context.CONTEXT_IGNORE_SECURITY);
```

La méthode *createPackageContext* de l'objet contexte, représenté par l'activité courante, renvoie au contexte de l'application dont le nom de package est passé en paramètre. À partir de cet objet, il est alors possible d'ouvrir des fichiers avec la méthode *openFileOutput* et *openFileInput* depuis l'espace de stockage de l'application en question.

En plus de nom de package, *createPackageContext* attend en paramètre un flag de type `int` `CONTEXT_IGNORE_SECURITY` ou `CONTEXT_INCLUDE_CODE`. Ce dernier indique que le *Context* retourné pourra servir à charger les classes de l'application mais on s'expose à lever *SecurityException*. Si on n'est pas intéressé par la

possibilité de charger les classes de l'application (par le `ClassLoader`), il est préférable de passer le flag `CONTEXT_IGNORE_SECURITY`.

En plus de la lecture des fichiers par `openFileInput` et de l'écriture par `openFileOutput`, le `Context` définit la méthode `deleteFile(String nomDuFichier)` pour supprimer les fichiers.

Enfin, pour clore le paragraphe « file system », on peut remarquer certaines méthodes de `Context` manipulant les fichiers au travers de la classe `java.io.File` :

- **getFilesDir()** renvoie l'objet `File` du répertoire racine de stockage. Son chemin est « `/data/data/{nom du package de l'application}/files` »
- **getCacheDir()** retourne quant à lui le répertoire où sont stockés les fichiers de cache : « `/data/data/{nom du package de l'application}/cache` ». Lorsque l'espace de stockage viendra à manquer, Android supprimera les fichiers présents dans ce répertoire.
- **getFileStreamPath(String nom_du_fichier)** retourne l'objet `File` qui correspond au fichier des méthodes `openFileInput` et `openFileOutput`.
- La méthode **getDir** sert à créer un répertoire propre à l'application mais qui n'est pas un sous-dossier du répertoire racine de stockage. Il se trouvera en dehors de l'espace habituel de stockage. Par exemple, pour une application dont le package est `org.florentgarin.android.persistance`, appeler cette méthode avec le nom « `monRepertoire` » en paramètre renverra le répertoire « `/data/data/org.florentgarin.android.persistance/app_monRepertoire` » et le créera s'il n'existe pas.

Travailler sur les fichiers à partir de l'objet `File` a l'avantage de permettre la création de sous-répertoires avec cette API Java standard. En effet, contrairement aux méthodes utilitaires de la classe `Context` (`openFileInput`, `openFileOutput`, `deleteFile`), il est possible de définir des structures arborescentes avec l'objet `File`.

6.3 LES PRÉFÉRENCES UTILISATEUR

Un autre procédé pour sauvegarder des données sur le mobile est le système des préférences. Il fonctionne un peu à la manière du package Java standard « `java.util.prefs` ». Il offre une API de plus haut niveau que la manipulation directe des flux de fichiers qui impose que l'on définisse son propre format. Au lieu donc de lire ou d'écrire des octets à travers un canal (stream), le système de préférence, géré par la classe `SharedPreferences`, lit et écrit des valeurs primitives (boolean, int, float, long, String).

Pour récupérer une instance de `SharedPreferences`, il existe deux méthodes : la méthode `getPreferences(int mode)` qui définit au niveau de la classe `Activity` et la méthode `getSharedPreferences(String name, int mode)` qui appartient à la classe `Context`. `Activity` étant également un `Context`, il sera possible d'invoquer les deux méthodes depuis le point d'entrée de l'application (`onCreate`).

La première méthode, *getPreferences*, retourne les préférences propres à l'activité, c'est pour cela qu'aucun nom n'est transmis en paramètre, chaque activité n'ayant qu'un unique ensemble de propriétés non partagé entre elles.

La méthode *getSharedPreferences* permet d'obtenir l'objet gérant l'ensemble des préférences regroupées sous le nom passé en paramètre. Ces deux méthodes attendent en paramètre un flag précisant le mode d'accès. Les valeurs possibles sont :

- `MODE_PRIVATE`
- `MODE_WORLD_READABLE`
- `MODE_WORLD_WRITEABLE`

Ces constantes ont la même signification que lorsqu'elles sont appliquées au système de fichiers, c'est-à-dire que les valeurs enregistrées avec un *SharedPreferences* en `MODE_PRIVATE` ne pourront être manipulées que par les applications tournant avec le même « user ID » alors que si le *SharedPreferences* utilisait le `MODE_WORLD_READABLE` ou le `MODE_WORLD_WRITEABLE`, ses données pourraient être lues ou modifiées pour toutes les applications.

6.3.1 Lecture des préférences

Les lectures des préférences se font assez simplement : une fois récupérée l'instance de *SharedPreferences* souhaitée, il suffit d'invoquer la méthode « get » correspondant au type de la valeur à chercher avec en paramètres sa clé et la valeur à renvoyer si elle n'est pas retrouvée (valeur par défaut).

L'exemple ci-dessous montre comment au démarrage de l'application on pourrait charger des éléments de paramétrage qui auraient été précédemment sauvegardés :

```
SharedPreferences settings = getPreferences(Context.MODE_PRIVATE);
boolean tvaReduite = settings.getBoolean("TVAReduite", false);
float prixBanane = settings.getFloat("prixBanane", 6.5F);
int stockDisponible = settings.getInt("stockDisponible", 500);
String etiquette = settings.getString("etiquette", null);
//affectation des valeurs
```

Dans l'exemple, le *SharedPreferences* chargé de stocker les données est privé, c'est celui dédié à l'activité. Si la méthode *getPreferences* était invoquée sur une autre activité, l'instance retournée serait différente et les valeurs ne seraient pas trouvées. C'est bien là la philosophie de cet objet ; son but est de mémoriser la configuration, les préférences, de l'activité.

6.3.2 Écriture des préférences

L'écriture des préférences a la particularité de se faire de manière atomique grâce à l'objet *SharedPreferences.Editor*. Une instance de cet objet est retournée par la méthode *edit()* de la classe *SharedPreferences*. C'est sur cette instance que seront appelées les méthodes *putBoolean*, *putFloat*, *putInt*, *putLong* et *putString* accompagnées de la clé et

de la valeur de la préférence. Ensuite, au moment de l'appel au `commit()`, les données positionnées par les appels aux méthodes « put » seront persistées toutes d'un seul bloc.

Voici un exemple d'utilisation :

```
SharedPreferences settings = getPreferences(Context.MODE_PRIVATE);
SharedPreferences.Editor editor = settings.edit();
editor.clear();
editor.putBoolean("TVAReduite", tvaReduite);
editor.putFloat("prixBanane", prixBanane);
editor.putInt("stockDisponible", stockDisponible);
editor.putString("etiquette", etiquette);
editor.commit();
```

La méthode `clear()` a pour effet d'effacer tous les enregistrements présents à l'instant où le `SharedPreferences.Editor` a été obtenu. Elle aurait pu être aussi appelée juste avant le `commit()`, le résultat produit aurait été identique, l'instance de `SharedPreferences` ne contiendrait *in fine* que les quatre valeurs mémorisées par l'objet `editor`.

Les modifications apportées à un `SharedPreferences` peuvent être suivies par le biais de l'enregistrement d'un callback, grâce à la méthode `registerOnSharedPreferenceChangeListener(SharedPreferences.OnSharedPreferenceChangeListener listener)`, la méthode `unregister...` sert à déréférencer ce callback.

6.3.3 IHM de Configuration

Même si le système des préférences peut être employé pour sauvegarder tout type de données primitives, il a avant tout été pensé pour enregistrer la configuration des applications. Le package « `android.preference` » contient un ensemble de classes qui aident à la création d'interfaces graphiques permettant d'afficher et d'éditer des préférences.

Grâce à ce framework, la création des écrans de configuration peut se faire au travers d'un fichier xml. Jusqu'à présent, rien de nouveau car c'est le cas pour tous les écrans avec les fichiers de layout, mais l'avantage d'utiliser le framework de préférence réside dans la possibilité de définir le « data binding » également dans le fichier xml. Ainsi la synchronisation entre l'interface visuelle de configuration et le fichier de stockage des préférences est automatique.

Le fichier de préférence doit être positionné dans le répertoire « `res/xml` ». Par exemple, voici le contenu du fichier `preferences.xml` :

```
<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android">
    <CheckBoxPreference
        android:key="tvaReduite"
        android:title="TVA à taux réduit ?"
        android:summary="5,5% ou 19,6%" />
    <EditTextPreference
        android:key="etiquette"
        android:title="Libellé de l'étiquette"
```

```

        android:summary="Sera affiché sur l'étiquette" />
    <RingtonePreference
        android:key="sonnerie"
        android:title="Sonnerie"
        android:showDefault="true"
        android:showSilent="true"
        android:summary="Choisissez une sonnerie" />
</PreferenceScreen>

```

Ce fichier déclare trois préférences dont les clés sont « tvaReduite », « étiquette » et « sonnerie ».

Ensuite, il faut charger ce fichier, un peu comme un layout classique. Pour cela, il existe une activité spéciale, *PreferenceActivity*, qu'il s'agit d'étendre :

```

package org.florentgarin.android.persistance;
import android.os.Bundle;
import android.preference.PreferenceActivity;
public class PreferencesUIFrameworkActivity extends PreferenceActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.preferences);
    }
}

```

Et voilà le travail :



Figure 6.1 — PreferenceActivity

Dans, l'exemple, on a utilisé un composant particulier qui est le *RingtonePreference* grâce auquel on peut choisir une sonnerie parmi celles disponibles.

Le framework permet aussi de regrouper, pour une meilleure ergonomie, certaines préférences au sein de *PreferenceCategory* et même aussi de les étaler sur plusieurs écrans.

Une fois ces préférences enregistrées, pour y accéder, il faut recourir à l'objet *PreferenceManager* :

```

SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(this);
//lecture des paramètres...

```

6.4 SQLITE

Si le besoin de stockage devient plus complexe, que les données à sauvegarder ne peuvent pas se modéliser aussi simplement que sous la forme d'une paire clé/valeur, il faudra envisager d'utiliser une véritablement base de données.

Android s'est doté d'une base de données spécialement conçue pour être suffisamment légère pour tourner sur des environnements aux capacités réduites tels que les téléphones portables. Cette base de données, c'est SQLite. Pour interagir avec cette base, les classes à utiliser se trouvent dans : « android.database » et « android.database.sqlite ».

Si l'on regarde la javadoc Android, on s'apercevra que les packages « java.sql » et « javax.sql » sont bien présents mais les classes de ces packages servent à communiquer au travers de l'API JDBC avec une base de données distante résidant sur un serveur. Ce n'est pas ce que l'on cherche à faire ici ; l'objectif est de persister des données localement sur le terminal, il faudra donc se concentrer sur les classes android.database.*.

Pour définir la base de données, la première étape consiste à fournir une implémentation concrète de la classe *SQLiteOpenHelper*, notamment les deux méthodes abstraites *onCreate* et *onUpgrade*. La méthode *onCreate* sera appelée automatiquement lorsque la base de données sera accédée pour la première fois. Il faut bien se souvenir qu'on n'est pas dans le contexte d'une application serveur d'entreprise où il est possible de configurer préalablement à loisir l'environnement d'exécution du programme. Une fois packagées, l'application Android se doit de pouvoir créer son environnement, y compris donc sa base de données.

Voici un cas d'exemple d'implémentation de *SQLiteOpenHelper* :

```
package org.florentgarin.android.persistance;
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
public class DBHelper extends SQLiteOpenHelper {
    private static final String DATABASE_NAME = "demo.db";
    private static final int DATABASE_VERSION = 1;
    private static final String CREATE_SQL = "CREATE TABLE STOCK ("
        + "CODE INTEGER PRIMARY KEY,"
        + "LIBELLE TEXT,"
        + "TARIF REAL,"
        + "DATE_CONSOMMATION INTEGER"
        + ");";
    private static final String DROP_SQL = "DROP TABLE IF EXISTS STOCK;";
    public DBHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }
    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(CREATE_SQL);
    }
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL(DROP_SQL);
    }
}
```



```

        onCreate(db);
    }
}

```

La méthode *onUpgrade* relève de la même philosophie que *onCreate* sauf qu'elle est invoquée lorsque la base de données déjà existante est d'une version antérieure à celle dorénavant requise.

Cette capacité qu'Android a de tracer les versions des bases de données au moment de leur création est très intéressante ; ainsi, si le schéma de la base est amené à évoluer lors du développement d'une version supérieure de l'application, il sera possible d'embarquer directement dans le code les scripts de migration.

Dans l'exemple précédent, un simple « drop and create » a été fait, mais si on devait gérer une vraie politique de migration, on devrait plutôt exécuter les scripts de type ALTER en fonction de l'ancienne version (*oldVersion*) encore installée.

Maintenant, à partir de cette instance de *SQLiteOpenHelper*, il faut récupérer un objet de type *SQLiteDatabase*. Pour ça, il y a deux méthodes :

- *getWritableDatabase*
- *getReadableDatabase*

La première est à appeler si l'on a l'intention de réaliser des modifications dans la base. La seconde seulement si on ne compte faire que des lectures.

Après, l'insertion d'un enregistrement se fait comme ceci :

```

SQLiteOpenHelper dbHelper = new DBHelper(this);
SQLiteDatabase db = dbHelper.getWritableDatabase();
ContentValues values = new ContentValues();
values.put("CODE", 2345);
values.put("LIBELLE", "bananes");
values.put("TARIF", 7.75F);
Calendar rightNow = Calendar.getInstance();
rightNow.set(Calendar.YEAR, 2011);
values.put("DATE_CONSOMMATION", rightNow.getTimeInMillis());
db.insert("STOCK", "LIBELLE", values);

```

Cet exemple fait l'usage de l'objet *ContentValues* pour encapsuler les données. *SQLiteDatabase* propose néanmoins l'exécution de banales, mais efficaces, requêtes sql en plus de cette API à la sauce objet.

Les transactions sont également supportées, les méthodes *beginTransaction()* et *endTransaction()* permettent d'en définir les démarcations.

Pour vérifier l'insertion effective dans la table, il est pratique de se connecter à la base grâce la commande *sqlite3* tapée dans une session adb (« adb shell ») (figure 6.2).

Le chemin à la base « /data/data/org.florentgarin.android.persistance/databases/demo.db » indique bien qu'à l'instar des fichiers de stockage, la base de données est propre à l'application et donc non partageable entre programmes. À moins bien sûr de l'encapsuler dans un *ContentProvider* pour exposer explicitement ses données.

```
# sqlite3 /data/data/org.florentgarin.android.persistence/databases/demo.db
sqlite3 /data/data/org.florentgarin.android.persistence/databases/demo.db
SQLite version 3.5.9
Enter ".help" for instructions
sqlite> select * from stock;
select * from stock;
2345|bananes|7.75|1384883717282
sqlite> _
```

Figure 6.2

6.5 EXPOSER SES DONNÉES

Comme expliqué aux paragraphes précédents, les données gérées sous forme de fichiers, de préférences ou de bases de données résident dans un espace privé à l'application les ayant créées.

La solution pour pouvoir partager des données entre applications est le *Content-Provider*. Une application qui souhaite ouvrir l'accès à ses données doit fournir une implémentation de cette classe abstraite. Cette dernière délivre des portes d'entrées pour rechercher, lire, écrire ou modifier les données. Le stockage physique des informations est parfaitement encapsulé et n'a aucune importance, il peut être de n'importe quelle nature ; il pourra d'ailleurs être aisément bouchonné.

Bouchon

Un bouchon est un code qui n'effectue aucun traitement. Les méthodes des classes bouchonnées renvoient invariablement le même résultat. Le bouchon sert d'alternative temporaire à un code dont le développement n'est pas encore stabilisé.

6.5.1 Utiliser les providers

Le mode de fonctionnement des providers est basé sur le paradigme REST (*REpresentational State Transfer*). Effet, les éléments sont identifiés par des URI (*Uniform Resource Identifier*) et les providers ne sont pas accédés directement mais par un objet de type *ContentResolver*.

Le rôle du *ContentResolver* est d'identifier le bon *ContentProvider* en fonction de l'URI qui lui est passé en paramètre ; les méthodes du *ContentResolver* pour manipuler les objets stockés, que ce soit en lecture ou en écriture, acceptent toutes une URI en paramètre.

Les données sont donc localisées par leurs URI : pour pouvoir utiliser les providers, il n'est pas nécessaire de savoir exactement quelle est la classe d'implémentation du *ContentProvider*, par contre il est primordial de connaître la typologie des URI.

Les URI sont de la forme suivante :

■ content://{authority}/{path}/{ID}

par exemple :

■ content://contacts/people/obama

« content » est dans la terminologie URI le *schema*. Par exemple, « http » est un autre *schema* très fréquemment utilisé pour construire des URIs¹. Ici « content » renseigne sur le fait que la ressource sera délivrée par un *content provider*.

Les termes « authority » et « path » proviennent aussi de la spécification des URI par la RFC 2396. « authority » désigne l'entité garante de la ressource pointée par l'URI.

Dans le cas présent, il s'agit du *content provider* ou plus exactement de la valeur définie comme telle dans le fichier manifeste de l'application qui déploie le *provider*. Il faut bien veiller à l'unicité de l'« authority », c'est pour cela qu'il est conseillé de choisir pour cette valeur le nom qualifié, tout en minuscule, de la classe du *provider*. Dans l'exemple plus haut, l'authority est « contacts », ce qui ne correspondant pas au nom de package du *provider*. Ce choix a sans doute été fait par Google pour raccourcir les URI d'accès à ce *provider* standard d'Android.

Ensuite, vient le terme « people ». C'est la partie « path » de l'URI. Cette partie identifie le type de la ressource. Toujours dans l'exemple, la valeur de path est « people », ce qui informe que le *provider* renverra une ressource encapsulant les informations d'une personne. Le « path » peut néanmoins être composé de plusieurs éléments séparés par un « / » s'il y a un besoin de hiérarchisation, on pourrait par exemple imaginer avoir « people/sales », « people/marketing »...

Enfin, en dernier lieu, et de façon optionnelle, on peut trouver l'identifiant unique de la ressource. Dans ce cas, un seul élément sera alors retourné (si bien sûr il existe). Si l'id n'est pas précisé, tous les éléments correspondant au path seront renvoyés.

Les requêtes se lancent par la méthode *query* du *ContentResolver*, l'objet retourné est un *Cursor* qui, tel un curseur de base de données, navigue entre les enregistrements et offre des méthodes permettant d'accéder aux champs en fonction de leur type (boolean, short, int, long, float, double, String, Blob).

En dépit du fait que la classe servant à lire les données du *provider*, la classe *Cursor*, soit la même que celle utilisée par l'API SQLite pour parcourir les résultats des requêtes SQL, il n'y a aucune obligation d'employer une base de données comme système de stockage sous-jacent aux données exposées par le *content provider*. Par contre, en raison de l'usage commun de la classe *Cursor*, ce choix paraît naturel.

```
package org.florentgarin.android.persistence;
import android.app.Activity;
import android.content.ContentResolver;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.provider.Contacts;
public class ContentProviderActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
```

1. Les URL (*Uniform Resource Locator*) sont avec les URN (*Uniform Resource Name*) les deux sous catégories d'URI. Contrairement aux URL, les URN (par exemple <mailto:info@docdoku.com>) ne font que nommer les ressources sans donner d'indication sur la façon de les atteindre.

```

public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ContentResolver cr = getContentResolver();
    Uri uri = Contacts.People.CONTENT_URI;
    Cursor cursor = cr.query(uri, null, null, null, null);
    if (cursor.moveToFirst()) {
        String name;
        String phoneNumber;
        int nameColumn =
cursor.getColumnIndexOrThrow(Contacts.People.NAME);
        int phoneColumn =
cursor.getColumnIndexOrThrow(Contacts.People.NUMBER);
        do {
            name = cursor.getString(nameColumn);
            phoneNumber = cursor.getString(phoneColumn);
            Log.i("ContentProviderActivity", "nom : " + name);
            Log.i("ContentProviderActivity", "numéro : " + phoneNumber);
        } while (cursor.moveToNext());
    }
    cursor.close();
}
}

```

Pour obtenir le *Cursor*, la méthode *query* est appelée. En plus de l'URI de la ou des ressources recherchées, qui est le premier paramètre, cette méthode permet de passer la projection, qui est la liste des colonnes à retourner, présentée sous forme de tableau de *String*. La sélection est une *String* représentant la clause *WHERE* (sans contenir le mot-clé *WHERE*) de la requête avec potentiellement des arguments matérialisés par un « ? », un tableau de *String* avec les valeurs de ces éventuels arguments et enfin la clause *SORTED BY* de la requête.

Ces paramètres sont tous optionnels et peuvent être remplacés par « null » : c'est le cas de l'exemple qui retourne toutes les colonnes de toutes les personnes du carnet d'adresses dans l'ordre par défaut.

Si on avait voulu, par exemple, ne récupérer que les colonnes « name », « number » et « notes » de toutes les personnes dont le nom commence par « J » triées dans l'ordre de leur nom, on aurait appelé *query* avec les paramètres suivants :

```

Cursor cursor = cr.query(uri, new String[] { Contacts.People.NAME,
Contacts.People.NUMBER, Contacts.People.NOTES },
Contacts.People.NAME + " LIKE ?", new String[] { "J%" },
Contacts.People.NAME);

```

Le reste de l'exemple est assez classique et ressemble fortement au fonctionnement du *ResultSet* de l'API *JDBC* ; les colonnes sont lues une à une en appelant la bonne méthode *get*.

Attention cependant, l'application nécessite pour pouvoir s'exécuter d'avoir la permission de lire les entrées du carnet d'adresses, sinon l'exception suivante sera remontée :

```
java.lang.RuntimeException: Unable to start activity ComponentInfo{org.florentgarin.android.persistence/org.florentgarin.android.persistence.ContentProviderActivity}: java.lang.SecurityException: Permission Denial: reading com.android.providers.contacts.ContactsProvider uri content://contacts/people from pid=246, uid=10020 requires android.permission.READ_CONTACTS
```

Pour accorder la permission `READ_CONTACTS` à l'application, il faut éditer le manifeste et y ajouter la ligne suivante juste avant la balise fermante du tag « manifest » :

```
<uses-permission
android:name="android.permission.READ_CONTACTS"></uses-permission>
```

Cette permission peut aussi s'ajouter à l'aide du *Plug-in* Eclipse d'Android. Bien sûr, pour voir s'afficher dans les logs le nom et le numéro des contacts, il faut en avoir saisi quelques-unes commençant par la lettre J.

6.5.2 Modifier les données du content provider

Les données des content providers peuvent, en plus d'être lues, être également modifiées, effacées, et il est aussi possible de rajouter de nouveaux enregistrements. Ces opérations se font, comme pour la lecture, au travers de l'objet *ContentResolver*.

Pour ajouter une entrée, il faut appeler la méthode `insert` en passant au *ContentResolver* l'URI de la ressource à insérer avec l'objet *ContentValues*, qui est une map où les clés sont les noms des colonnes indexant les valeurs de l'enregistrement.

Par exemple, les lignes suivantes ajoutent un contact au carnet d'adresse :

```
ContentValues values = new ContentValues();
values.put(Contacts.People.NAME, "Jessica");
values.put(Contacts.People.NOTES, "Elle est très sympa");
Uri uri = cr.insert(Contacts.People.CONTENT_URI, values);
Log.i("ContentProviderActivity", " URI : " + uri.toString());
return uri;
```

Là aussi, une permission est à ajouter à l'application pour qu'elle puisse s'exécuter : c'est celle autorisant l'écriture dans les contacts : « `android.permission.WRITE_CONTACTS` ».

L'URI retournée par le content provider est l'URI pointant vers l'entrée venant d'être enregistrée : « `content://contacts/people/8` ». On voit bien ici que l'id est un entier qui s'incrémente à chaque nouvel enregistrement. Si l'application est lancée plusieurs fois, autant de contacts portant le même nom seront ainsi créés.

Dans l'émulateur, pour voir apparaître ces nouveaux contacts, il faut modifier le filtre d'affichage des entrées du répertoire. Pour cela, il faut cliquer sur le bouton « Display group » du menu général puis changer la sélection de « My Contacts » à « All contacts ».

Grâce à l'URI du contact, il est ensuite aisé de créer d'autres sous-éléments liés. Dans le contexte du carnet d'adresse qui est celui de l'exemple, un contact peut avoir plusieurs numéros de téléphones (mobile, bureau, maison...) ou plusieurs emails. Ces objets sont des ressources elles-mêmes pointées par des URI qui reflètent bien la hiérarchisation de l'information.

L'exemple précédent pourrait se prolonger par l'insertion d'un numéro de téléphone à la fiche contact venant d'être créée :

```
Uri phoneUri = Uri.withAppendedPath(uri,
Contacts.People.Phones.CONTENT_DIRECTORY);
values.clear();
values.put(Contacts.People.Phones.TYPE, Contacts.People.Phones.TYPE_MOBILE);
values.put(Contacts.People.Phones.NUMBER, "0561234567");
cr.insert(phoneUri, values);
```

La valeur de phoneUri est « content://contacts/people/8/phones » : cet objet est construit en ajoutant à l'URI du contact la constante CONTENT_DIRECTORY « phones ». Après, l'insertion se fait comme précédemment, on peut remarquer que l'objet *ContentValues* est réutilisable.

En plus de l'opération *insert*, *ContentResolver* propose les méthodes *delete* et *update* pour supprimer et mettre à jour des enregistrements. Le fonctionnement de ces méthodes repose sur une URI passée en paramètre et la définition optionnelle de la clause WHERE précisant les éléments auxquels s'applique le traitement.

6.5.3 Créer son propre ContentProvider

Pour créer son propre *ContentProvider* et non plus seulement utiliser ceux mis à disposition par les autres applications, il faut étendre la classe *ContentProvider* et proposer une implémentation pour les méthodes :

- delete
- getType
- insert
- query
- update

L'application NotePad incluse dans le sdk Android donne un bon exemple de *ContentProvider*. Lorsque le système de stockage est SQLite3, la stratégie d'implémentation peut se résumer à analyser l'URI fournie en paramètre puis à déléguer l'exécution de la méthode à l'instance de la classe *SQLiteDatabase*.

Pour faciliter la construction du *ContentProvider* et prendre en charge les tâches répétitives, le sdk contient quelques classes utilitaires. La classe *UriMatcher* permet de déterminer la nature de l'URI : c'est-à-dire le type de contenu et s'il s'agit de l'URI pointant vers la collection de ressources ou d'un élément en particulier.

Le principe d'*UriMatcher* est le suivant, les patterns d'URI sont affectés à une constante de type int (NOTES et NOTE_ID dans l'exemple), le symbole '#' représentant l'id de type entier :

```
sUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
sUriMatcher.addURI(NotePad.AUTHORITY, "notes", NOTES);
sUriMatcher.addURI(NotePad.AUTHORITY, "notes/#", NOTE_ID);
```

Ensuite, la méthode *match* renvoie la constante correspondant au pattern de l'URI en paramètre :

```
switch (sUriMatcher.match(uri)) {
    case NOTES:
    ...
```

La méthode *getType* est la seule ne débouchant pas sur une requête dans la base. Son but est de renvoyer le type MIME de la donnée pointée par l'URI. La règle est la suivante : pour les données individuellement identifiées, le type MIME est :

```
vnd.android.cursor.item/vnd.{nom de l'organisation responsable de
l'application}.{le type de contenu}
```

Et pour l'URI collective :

```
vnd.android.cursor.dir/vnd.{nom de l'organisation responsable de
l'application}.{le type de contenu}
```

Ainsi pour l'application *NotePad*, nous avons :

```
vnd.android.cursor.item/vnd.google.note
```

et

```
vnd.android.cursor.dir/vnd.google.note .
```

Le curseur délivré par les API SQLite sera bien entendu celui renvoyé par la méthode *query* du *ContentProvider*. Cependant, celui-ci se doit de disposer d'une colonne « *_id* » de type int contenant l'identifiant de la ressource. Il faut donc bien penser à créer la table avec comme clé primaire cette colonne (BaseColumns._ID est une constante équivalent à « *_id* »).

Au-delà de ces directives obligatoires, il ne faut pas oublier qu'un content provider est une API externe qui sera utilisée par des applications tierces. Par conséquent, il est plus que conseillé de bien documenter le type des données, le nom des tables, des colonnes par l'ajout de constantes ; les URI aussi doivent être bien connues, la convention veut que la constante « *CONTENT_URI* » référence l'URI racine du provider.

Les données binaires

Les content providers supportent la gestion des données binaires de deux façons : soit le contenu est stocké directement dans la base de données dans une colonne *Blob*, soit c'est une simple référence vers le fichier binaire qui est enregistré dans la base.

Le choix de l'une et l'autre des stratégies de stockage répond au même critère que pour les applications serveurs classiques : tant que la taille des données (fichier image, vidéo, son...) est réduite, il est commode de choisir le *Blob*. Le fichier binaire est alors considéré comme n'importe quelle autre colonne et peut ainsi participer aux transactions, être sauvegardé en même temps que la BD...

Par contre, lorsque le fichier est plus gros, les inconvénients commencent à prendre le dessus sur les avantages : en effet une colonne *Blob* volumineuse est mal adaptée à la plupart des traitements d'une base de données comme la journalisation ou la gestion des rollbacks. Dans ce cas, il vaut mieux sortir le fichier de la base pour le mettre directement sur le filesystem.

Sur Android, pour les petits fichiers, la colonne accueillant les données sera du type SQL *Blob* et au niveau du code, elle sera manipulée au travers d'un tableau d'octet (`byte[]`). L'objet *Cursor* définit la méthode *getBlob* qui renvoie ce tableau.

Pour les gros fichiers, Android bénéficie d'un mécanisme assurant de manière transparente le lien entre l'enregistrement en base et le fichier physique. L'astuce est de déclarer une colonne spéciale nommée « *_data* » qui contiendra le chemin vers le fichier. Cette colonne n'est pas prévue pour être lue et écrite manuellement : c'est au *ContentResolver* de maintenir cette colonne.

Pour obtenir le stream d'écriture dans le fichier, sans devoir se soucier de son emplacement, il suffit de taper :

```
OutputStream out = cr.openOutputStream(uri);
```

À lecture, c'est pareil, il n'est pas nécessaire de connaître l'emplacement du fichier :

```
InputStream in = cr.openInputStream(uri);
```

Lors d'une lecture, le *ContentResolver* se charge de récupérer le chemin du fichier dans la colonne « *_data* », de l'ouvrir puis de retourner l'objet *InputStream* ; dans le cas d'une écriture, il créera le fichier, stockera son chemin dans la colonne « *_data* » et renverra l'*OutputStream*.

Déclaration dans le manifeste

La dernière étape pour achever la création de son propre fournisseur de contenu est la déclaration de celui-ci dans le fichier manifeste. Cette étape est essentielle car c'est elle qui rend publique l'existence du provider autorisant les invocations des autres applications.

La déclaration minimale est la suivante :

```
<provider
  android:name="org.florentgarin.android.persistence.ExampleContentProvider"
  android:authorities="org.florentgarin.android.persistence.examplecontentprovider">
</provider>
```

On précise ici le nom de la classe qualifiée et l'autorité de l'URI du provider. En plus de cela, on peut préciser les permissions qui seront requises pour lire ou écrire dans le provider, d'autres paramètres permettent aussi de contrôler finement les aspects processus. Par défaut, un provider est un composant instancié dans le même processus (même VM) que l'application où il est défini. Les requêtes émanant d'autres applications, tournant dans d'autres processus, passent alors par des mécanismes IPC (*Inter-Process Communication*) générant un coût additionnel sur les performances. Si l'attribut « android:multiprocess » est positionné à la valeur « true », Androidinstanciera le provider directement dans chacun des processus client, il n'y aura ainsi plus d'overhead. Cependant, le provider pourra potentiellement être instancié plusieurs fois ; sa conception doit donc le prévoir.

Ce qu'il faut retenir

Les possibilités d'Android en matière de persistance de données sont nombreuses : système de fichiers, base de données, préférences ou pourquoi pas chez les autres applications s'il en existe une particulièrement adaptée à gérer le type de données concerné.

Dans la plupart des cas, la base de données devrait être le choix à retenir, à part pour les informations de préférences utilisateur.

Enfin, une dernière possibilité, non abordée dans ce chapitre, serait de déporter les données sur un serveur distant, sur le « cloud » !

7

Fonctions IHM poussées

Objectifs

Les widgets et les layouts vus précédemment sont les briques de base de la construction d'interfaces. Cependant, Android, dans son kit de développement, propose encore beaucoup plus : des skins, des animations, des API 2D et 3D, un système de gadgets équivalent mobile du Dashboard de MacOS...

Ce chapitre tâchera de lever le voile sur ces éléments, grâce auxquels on peut faire des applications encore plus abouties visuellement.

7.1 LES MENUS

Il existe deux catégories de menus distinctes :

- les « *Option Menus* »,
- les « *Context Menus* ».

Ces menus sont différents en de nombreux points ; ils ne se déclenchent pas suite au même événement, n'ont pas un visuel identique, ont leur propre règle de fonctionnement. Sur le plan ergonomique, ils ne servent pas non plus le même type d'interaction utilisateur.

Cependant, ils ont au moins un point commun, comme les layouts : ils peuvent tous les deux être définis dans le code de l'application ou par un fichier xml.

7.1.1 Les « Option Menus »

Les « *Option Menus* » sont les menus principaux de l'application. Ils sont rattachés aux activités qui ne peuvent en définir qu'un seul. Ils apparaissent quand on clique sur le bouton « MENU » du téléphone. Chaque entrée du menu peut être accompagnée d'une icône. Par contre, ces menus ne pourront pas contenir de case à cocher ou de bouton radio. Ces menus sont destinés à présenter une liste d'actions ou à fournir des choix de navigation.

Pour créer un menu « Option », il faut redéfinir la méthode *public boolean onCreateOptionsMenu(Menu menu)* sur l'activité.

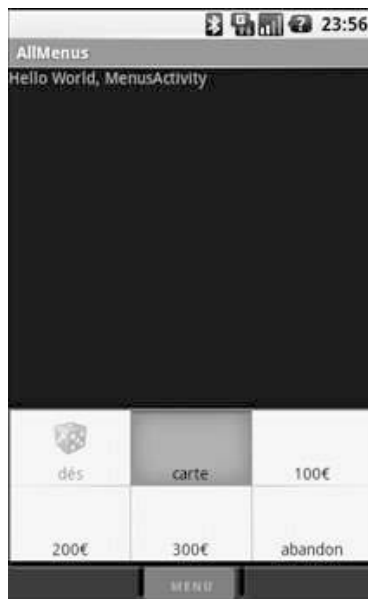


Figure 7.1 — Menu principal de l'activité

Pour obtenir le résultat ci-dessus, on peut soit créer le menu par du code java dans la méthode *onOptionsItemSelected* soit le déclarer en xml de la façon suivante :

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"><item
  android:id="@+id/item01" android:title="Lancer les dés"
  android:titleCondensed="dés" android:icon="@drawable/die"
  android:enabled="false"></item>
<item android:id="@+id/item02" android:title="Prendre une carte"
  android:titleCondensed="carte"></item>
<item android:id="@+id/item03" android:title="Parier 100 €"
  android:titleCondensed="100 €"></item>
<item android:id="@+id/item04" android:title="Parier 200 €"
  android:titleCondensed="200 €"></item>
<item android:id="@+id/item05" android:title="Parier 300 €"
  android:titleCondensed="300 €"></item>
<item android:id="@+id/item06" android:title="Abandonner la partie"
  android:titleCondensed="abandon"></item>
</menu>
```

Puis l'instancier, toujours dans la méthode « callback » `onCreateOptionsMenu` :

```
package org.florentgarin.android.menus;
import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuInflater;
public class MenusActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
    public boolean onCreateOptionsMenu(Menu menu) {
        MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.main_menu, menu);
        return true;
    }
}
```

Pour rattacher à chacun des items de ce menu à des actions, il y a deux possibilités :

- enregistrer un listener de type `MenuItem.OnMenuItemClickListener` sur le `MenuItem` ;
- redéfinir la méthode `onOptionsItemSelected` sur l'activité :

```
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.item01:
            //l'item 1 a été sélectionné
            return true;
        case R.id.item02:
            //l'item 2 a été sélectionné
            return true;
        case R.id.item03:
            //l'item 3 a été sélectionné
            return true;
        case R.id.item04:
            //l'item 4 a été sélectionné
            return true;
        case R.id.item05:
            //l'item 5 a été sélectionné
            return true;
    }
    return false;
}
```

Cette méthode est invoquée par le système lorsqu'un élément du menu est sélectionné. Il est probablement plus commode de définir les actions par ce biais-là plutôt que par l'enregistrement de listeners car, à l'aide d'un simple switch, on a une vision d'ensemble des commandes du menu.

Cette méthode déclare pour valeur de retour un boolean. Celui-ci indique si l'événement a été consommé, c'est-à-dire entièrement traité ou pas. Si on retourne false, l'éventuelle listener écoutant l'item choisi sera à son tour appelé. Si on retourne true, la propagation ne sera pas faite.

7.1.2 Les « Context Menus »

Les menus contextuels ont un fonctionnement très proche des options menu. Ils ne sont néanmoins pas liés à une activité mais à un objet de type *View*. Ce sont les équivalents de menus qui apparaissent après un clic droit sur un élément graphique des applications traditionnelles de bureau.

Sur un même écran, plusieurs menus contextuels pourront donc être enregistrés ce qui explique que, contrairement aux menus propres aux activités, ces menus ne supportent pas les raccourcis clavier ni d'ailleurs les icônes.

Le clic droit n'existant pas sur un système tactile, pour déclencher les menus contextuels, il faut maintenir la pression du clic quelques instants.

Pour créer un menu contextuel sur un widget donné, il faut enregistrer un listener de type *View.OnCreateContextMenuListener* sur celui-ci. La classe *Activity* implémente cette interface, il est donc plus facile de s'en servir que de créer un nouveau listener. La méthode *registerForContextMenu(View view)* présente sur la classe *Activity* s'enregistre elle-même sur l'objet *View* passé en paramètre. Il faudra donc invoquer cette méthode pour tous les widgets qui devront posséder un menu contextuel.

Ensuite, il convient bien sûr d'implémenter l'unique méthode du listener *onCreateContextMenu(ContextMenu menu, View v, ContextMenuInfo menuInfo)* sur l'activité. La logique est la même que pour la méthode *onCreateOptionsMenu*, le menu fourni en paramètre est celui qui sera rattaché à l'objet *View v* et qu'il s'agit de construire soit par du code soit à partir d'une définition XML. L'opération de rattachement au widget est automatique, il n'est pas nécessaire de faire quoique ce soit en ce sens sur le paramètre *v*. Ce paramètre est là afin que l'on sache quel est le composant pour lequel on est en train de créer le menu.

Enfin, la réception des événements de sélection dans le menu se fait en redéfinissant la méthode *onContextItemSelected(MenuItem item)* sur l'activité. Exactement comme pour la méthode *onOptionsItemSelected*, il est possible de faire un switch sur l'item id et de déclencher ainsi l'action correspondant à la valeur choisie par l'utilisateur dans le menu contextuel.

7.2 ÉTENDRE LES COMPOSANTS EXISTANTS

Lorsque les widgets présents nativement ne correspondent pas exactement à ce que l'on recherche, il est toujours possible d'en créer de nouveaux qui répondent entièrement au besoin. Pour concevoir son propre composant graphique, plusieurs stratégies sont possibles en fonction du degré de customisation.

L'approche la plus simple et la plus légère consiste à créer une nouvelle classe qui étend le widget à partir duquel on souhaite partir pour concevoir le nouveau composant, puis à simplement redéfinir la ou les méthodes commençant par « on » comme *onDraw*, *onMeasure*...

Ces méthodes sont les méthodes dites « callback methods » qui sont appelées par le système lorsque celui requiert une information ou une action de la part du widget.

Une variante de cette approche qui, en général, donne satisfaction est de créer un nouveau composant en partant d'un layout. Ensuite, on procède par agrégation d'autres widgets. L'exemple classique est le combobox qui réunit une zone de texte, un bouton et une liste déroulante.

L'essentiel est de bien encapsuler cette association de widgets élémentaires afin d'obtenir un nouveau composant qui sera traité de façon unitaire. Dans le fichier de layout XML, on y fera référence comme ceci :

```
<view  
    class="nom_qualifié_de_la_classe" ...
```

ou

```
<nom_qualifié_de_la_classe  
    id="@+id/xxx"  
    ... />
```

Enfin, une dernière méthode beaucoup plus radicale consiste à partir d'une feuille blanche ou presque en étendant l'objet widget de base : la classe *View*. Là, tout sera à faire : il faudra obligatoirement redéfinir la méthode *onDraw* pour dessiner le composant ainsi que la méthode *onMeasure* pour déterminer la taille de celui-ci.

Le composant pourra définir ses propres attributs XML qui seront récupérés dans le constructeur de l'objet grâce au paramètre de type *AttributeSet*. Le rôle de cette classe est de regrouper les attributs xml d'un tag pour une manipulation aisée.

Lorsque l'on crée son propre composant, il est primordial de bien le documenter. Par exemple, les attributs xml devront être recensés et bien expliqués dans la javadoc.

| Pour réaliser un widget en 3D, il faudra étendre la classe *SurfaceView* et non *View*.

7.3 LES ANIMATIONS

Au-delà de ses capacités 2D et 3D (voir quelques paragraphes plus loin dans ce même chapitre) qui autorisent la réalisation d'animations graphiques complexes, Android intègre un petit moteur de rendu permettant de faire des animations simples décrites soit sous forme XML soit avec du code. Comme pour la définition du layout de l'IHM, la forme XML est souvent plus lisible et facile à maintenir. Du point de vue pratique, ce fichier XML, qui est un fichier de ressources, doit être localisé dans le répertoire *res/anim*.

Les animations sont des assemblages d'effets élémentaires qui appliquent une transformation sur une instance d'objet *View*. La façon dont ces éléments de base s'enchaînent les uns après les autres en suivant une chronologie bien précise ou simultanément est décrite dans le fichier XML. À l'instar du layout de l'interface graphique, les animations peuvent toutefois se définir dans le code, le package « *android.view.animation* » contient les classes mises en jeu.

Il existe quatre animations de base :

- **alpha** - Cette animation réalise un effet de fondu en jouant sur le niveau de transparence.
- **scale** - L'animation est créée en effectuant un redimensionnement, les étapes intermédiaires étant visibles.
- **translate** - Effectue un mouvement de translation horizontale ou verticale ou les deux. Les coordonnées X, Y de la position de départ et ceux de la position d'arrivée sont les paramètres à fixer de l'animation.
- **rotate** - Il s'agit d'un mouvement de rotation. L'animation se configure en spécifiant les coordonnées du point de pivotage et l'angle de la rotation.

Les définitions des animations peuvent être regroupées entre elles par l'objet *AnimationSet* ou par le tag `<set>` si la déclaration se fait en XML. En les regroupant, on peut leur appliquer des propriétés communes, en particulier des propriétés se rapportant à l'organisation temporelle des animations. Ces attributs xml de la balise « set » ou leurs équivalents objets sont par exemple la « duration », qui fixe la durée en millisecondes de l'effet, « startOffset » qui, s'il est appliqué par contre individuellement aux animations, permet de décaler leur démarrage, `repeatCount` qui définit le nombre de fois que l'effet devra se répéter (la valeur -1 signifiant une répétition à l'infini)...

Parmi tous les attributs de gestion du temps, il y en a un très puissant qui mérite qu'on s'y attarde : il s'agit de la propriété `interpolator`.

Par cet attribut, on détermine l'objet de type *android.view.animation.Interpolator* qui gouvernera la manière dont l'effet s'appliquera à l'objet *View* dans le temps.

Par exemple, la classe *AccelerateInterpolator* lance l'animation doucement pour la terminer plus vite. *AccelerateDecelerateInterpolator* commence lentement, accélère ensuite pour finir à nouveau lentement. En dehors des quelques *Interpolator* du SDK, en plus de ceux déjà cités il faut ajouter *CycleInterpolator*, *DecelerateInterpolator* et *LinearInterpolator*, on peut toujours créer son propre *Interpolator* en implémentant l'interface du même nom.

Après avoir conçu une animation complète, il ne reste plus qu'à la lier à un objet *View* pour alors lancer son exécution.

Ci-dessous, une petite animation composée de deux effets, une rotation et une translation, toutes les deux démarrant et s'arrêtant de concert. L'animation sera appliquée à une image représentant un réveil pour simuler le mouvement qu'il ferait s'il venait à sonner :

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:shareInterpolator="true" android:interpolator="@anim/cycle">
    <translate android:fromXDelta="0" android:toXDelta="10"
        android:duration="10000" />
    <rotate android:fromDegrees="0" android:toDegrees="10"
        android:pivotX="24" android:pivotY="24" android:duration="10000" />
</set>
```


Cette animation utilise un interpolator de type *CycleInterpolator*. Il est défini dans un autre fichier XML lui aussi stocké dans le répertoire *res/anim*. Cet interpolator accepte un unique paramètre qui est le nombre de fois que l'animation sera répétée. *CycleInterpolator* détermine la fréquence de l'animation en suivant une courbe sinusoïdale :

```
<?xml version="1.0" encoding="utf-8"?>
<cycleInterpolator xmlns:android="http://schemas.android.com/apk/res/android"
    android:cycles="50" />
```

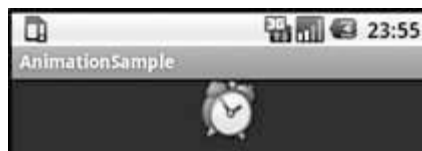
Ensuite, au niveau du code, on lit le fichier XML pour en retirer une instance de la classe *Animation* pour ensuite la rattacher à une *ImageView* :

```
package org.florentgarin.android.animation;
import android.app.Activity;
import android.os.Bundle;
import android.view.animation.Animation;
import android.view.animation.AnimationUtils;
import android.widget.ImageView;
public class AnimationSampleActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        ImageView alarmImage = (ImageView) findViewById(R.id.alarm);
        Animation ringingAnimation = AnimationUtils.loadAnimation(this,
R.anim.ring);
        alarmImage.startAnimation(ringingAnimation);
    }
}
```

Le réveil se met alors à vibrer :



Figure 7.2 — Le réveil...



...vibre !

La translation de quelques pixels, ajoutée à une rotation d'un angle faible, donne l'impression qu'il sonne.

Dans l'exemple, l'animation commence immédiatement. Il est cependant possible d'appliquer l'animation au widget, par la méthode *setAnimation*, puis de planifier son démarrage différé par la méthode *setStartTime*. Une autre classe, *LayoutAnimationController*, permet de faire des choses encore plus sophistiquées. L'objectif de cette classe est de contrôler le déclenchement d'une même animation sur plusieurs widgets d'un *ViewGroup*.

Certaines animations sont trop complexes pour pouvoir être créées simplement en faisant subir à un widget des déformations successives.

Android dispose d'un autre type d'animation appelée « Frame Animation ». Le principe en est simple, l'animation est construite en faisant défiler à l'écran une série de *Drawable* à l'instar des dessins animés traditionnels en 2D. Pour décrire ce genre d'animation, il suffit de lister les références vers les images dans le fichier XML avec pour chacune la durée d'affichage.

7.4 PERSONNALISATION EN FONCTION DE LA CONFIGURATION

Comme on le sait, Android n'est pas lié à un matériel particulier ni même à un constructeur. En outre, si Android a été initialement présenté comme un système mobile, il semble intéresser de plus en plus les fabricants de notebooks ou de tablettes tactiles. À terme, un nombre important d'appareils aux spécifications techniques hétérogènes seront motorisés par Android. Certains auront un large écran, d'autres seront dépourvus de la fonction tactile, et bien entendu la langue avec laquelle le système aura été configuré pourra varier... Une application Android pourra donc se retrouver installée sur un parc diversifié de terminaux. Il convient donc de concevoir l'application de telle sorte qu'elle puisse s'adapter au matériel pour fonctionner toujours de façon optimale.

7.4.1 Les ressources alternatives

Les fichiers ressource, étudiés en détail dans un chapitre au début de ce livre, sont tous stockés dans des sous-répertoires du dossier « res ». Un aspect intéressant du framework Android au sujet de la gestion des ressources est la possibilité de définir des fichiers ressources alternatifs. Le choix du fichier à retenir se fera par Android à l'exécution en fonction de certaines caractéristiques du système. Ce mécanisme est très proche de celui des *ResourceBundle* de Java où la JVM sélectionne le fichier properties dont le nom se termine par la langue et le pays (par exemple : *conf_fr_FR.properties*) correspondant au mieux au paramétrage courant.

Cependant, sur Android, les critères sur lesquels repose la sélection des ressources ne se limitent pas seulement aux éléments de localisation mais comprennent également des points techniques comme la résolution de l'affichage ou l'orientation de l'écran.

Le stockage des différents fichiers de ressources se fait dans des répertoires indépendants dont le nom suit une stricte codification : il s'agit du nom standard du répertoire hébergeant le type de la ressource auquel on a ajouté la ou les valeurs des caractéristiques de configurations exigées séparées par un tiret « - » et dans un ordre bien précis.

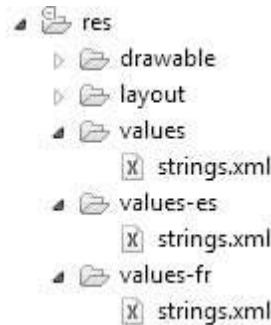


Figure 7.3 — Localisation du fichier strings.xml

Par exemple, pour réaliser une traduction des chaînes de caractères en français et en espagnol et conserver l'anglais comme la langue par défaut à utiliser si la langue configurée sur le téléphone n'est aucune des deux autres, il faudrait alors créer l'arborescence suivante.

Le fichier strings.xml sous le répertoire values-fr a le contenu suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Bienvenue</string>
    <string name="app_name">AlternateConfiguration</string>
</resources>
```

Celui sous values-es, celui-ci :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Bienvenido</string>
    <string name="app_name">AlternateConfiguration</string>
</resources>
```

Enfin, le fichier de ressource par défaut sous le répertoire values :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Welcome</string>
    <string name="app_name">AlternateConfiguration</string>
</resources>
```

À l'exécution, le libellé qui apparaîtra à l'écran sera déterminé en fonction de la configuration courante de la langue (figures 7.4 à 7.6).

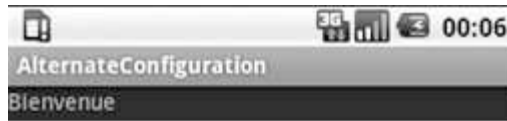


Figure 7.4 — Version française

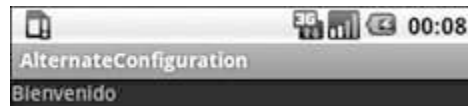


Figure 7.5 — Version espagnole

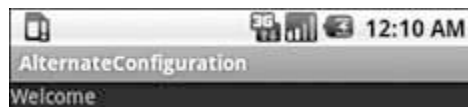


Figure 7.6 — Version internationale

Le code est très simple, le fichier layout main est chargé et positionné comme content view :

```
/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
}
```

Dans le fichier layout, la ressource est identifiée uniquement par son nom « @string/hello » sans aucune référence à la langue à utiliser ni au chemin du fichier physique qui devra être lu. C'est parfaitement logique vu que cette résolution est faite par Android automatiquement à l'exécution :

```
<?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:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello"
        />
</LinearLayout>
```

Il n'existe d'ailleurs aucune API pour spécifier la langue courante de l'application alors que dans une JVM classique, cela est possible grâce à la méthode `Locale.setDefault(locale newLocale)`.

La seule façon de choisir la langue de l'application est d'effectuer la sélection au niveau du système en lançant l'activité « Paramètres » puis en navigant dans le menu « Par. régionaux et texte/Langue et région ».

L'exemple précédent est classique mais il faut néanmoins bien comprendre que n'importe quelle ressource peut faire l'objet d'une localisation et ce quelle que soit sa nature. Par exemple, une ressource de type Color déclarée dans le fichier `strings.xml` pourrait très bien exister sous des formes alternatives dans les fichiers `values-fr/strings.xml` et `values-es/strings.xml`. On peut même aller plus loin en créant des versions localisées des fichiers `layout` qui seront stockés dans les dossiers `layout-es`, `layout-fr`...

Le choix de la langue n'est qu'un point de différenciation parmi d'autres. Voici la liste exhaustive et ordonnée de tous les critères pouvant intervenir dans la définition de ressources alternatives :

- **Langage** - Les deux lettres en minuscule de la langue selon la norme ISO 639-1 (en, fr, es). Ce critère est le plus important dans l'algorithme de résolution.
- **Région** - Il s'agit du code du pays. Ce code est en majuscule et doit être précédé par la lettre r. (rFR, rUS, rES).
- **L'orientation de l'écran** - Les valeurs possibles sont « port » pour portrait, « land » pour landscape, c'est-à-dire paysage, et « square » pour les écrans carrés.
- **La précision de l'écran** - Exprimée en dpi, dots per inch (point par pouce), par exemple 92 dpi ou 108 dpi.
- **Les caractéristiques tactiles de l'écran** - Les valeurs possibles sont « notouch » pour signifier que l'écran n'est pas tactile, « stylus » pour un écran répondant au stylet et « finger » pour ceux pouvant être manipulés directement avec le doigt.
- **Disponibilité du clavier** - « keysexposed » pour indiquer que le clavier est sorti et « keyshidden » pour dire qu'est caché.
- **Méthode de saisie de texte** - Trois valeurs sont possibles « nokeys » c'est-à-dire « pas de touche », « qwerty » qui veut dire que le téléphone dispose d'un véritable clavier ou « 12key » qui désigne les claviers, possédant 12 touches numériques (touches de 0 à 9 plus « * » et « # ») dont sont équipés la plupart des téléphones. Grâce ce critère, on peut imaginer une interface graphique spécialement conçue pour les téléphones non équipés de clavier complet pour faciliter la saisie, en présentant une sorte de carrousel où les lettres défileraient. Ce layout serait alors défini dans le répertoire « res/layout-12key ».
- **Méthode de navigation du téléphone autre que le tactile** - Il est aussi possible de créer des ressources personnalisées qui dépendent du périphérique de navigation dont le terminal est muni : « nonav » pour « no navigation », « dpad » pour directional pad, « trackball » ou « wheel ».

- **Les dimensions de l'écran** - Il faut ici préciser la résolution de l'écran. Par exemple, si l'on veut fournir une image différente (une ressource de type `drawable`) en fonction des dimensions de l'écran, on créera les répertoires « `res/drawable-320x240` » et « `res/drawable-640x480` » pour accueillir les fichiers images spécifiques.

Ces critères peuvent être ou ne pas être cumulés mais il est crucial qu'ils apparaissent dans l'ordre de la liste ci-dessus. Par exemple, pour une animation en français pour larges écrans, le fichier de ressource se retrouvera dans le répertoire « `res/anim-fr-640x480` ».

7.5 NOTIFIER L'UTILISATEUR

Sur une plateforme communicante telle qu'Android, il arrive fréquemment que l'on ait besoin d'avertir l'utilisateur qu'un événement vient de se produire.

Android dispose de trois modes de notifications, chacun étant adapté à la nature du message à faire passer.

7.5.1 Le Toast

Le *Toast* est un message flash qui apparaît de manière plus ou moins furtive mais non discrète à l'écran. Un toast peut être généré depuis une activité mais aussi depuis un service tournant en tâche de fond.

Le *Toast* est particulièrement indiqué pour transmettre à l'utilisateur des informations immédiates dont on veut qu'elles parviennent à l'utilisateur le plus tôt possible mais sans avoir besoin d'être sauvegardées. Par exemple, pour notifier la fin d'un téléchargement ou l'installation d'un composant, le *Toast* remplirait parfaitement son rôle.

Pour afficher un toast basique, c'est-à-dire uniquement un texte sans image ni style particulier, il suffit d'utiliser la méthode statique *makeText* :

```
Toast.makeText(this, "Yep voici mon toast", Toast.LENGTH_LONG).show();
```

pour voir apparaître l'encart suivant qui disparaîtra dans un effet de fondu :



Figure 7.7 — Le Toast : simple message de type popup

Il est également possible d'afficher un *Toast* plus original et personnalisé. Pour cela, il faut créer un layout comme un écran d'activité :

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/custom_toast"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="10dp"
    android:background="#DAAA"
    >
    <ImageView android:id="@+id/image"
        android:layout_width="wrap_content"
        android:layout_height="fill_parent"
        android:layout_marginRight="10dp"
        />
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="fill_parent"
        android:textColor="#FFF"
        />
</LinearLayout>
```

Ce layout définit une zone d'image et un texte alignés horizontalement. Ensuite, on obtient le layout dans le code grâce à l'objet *LayoutInflater* dont la responsabilité est de reconstruire un objet *View* à partir d'un fichier de définition XML.

Ce fichier layout n'étant qu'une sorte de template, il faut ensuite positionner l'image et le texte sur les widgets correspondants :

```
LayoutInflater inflater = getLayoutInflater();
View layout = inflater.inflate(R.layout.my_toast_layout,
    (ViewGroup) findViewById(R.id.custom_toast));
ImageView image = (ImageView) layout.findViewById(R.id.image);
image.setImageResource(R.drawable.logo);
TextView text = (TextView) layout.findViewById(R.id.text);
text.setText("Bienvenue dans un monde merveilleux !");
Toast toast = new Toast(this);
toast.setGravity(Gravity.CENTER, 0, -80);
toast.setDuration(Toast.LENGTH_LONG);
toast.setView(layout);
toast.show();
```

Avant son affichage, le toast est positionné à l'écran par la méthode ; aligné horizontalement et 80 pixels au-dessus du centre verticalement (figure 7.8).

On peut bien noter la transparence du toast qui ne masque pas complètement les éléments présents sur l'activité en arrière-plan.

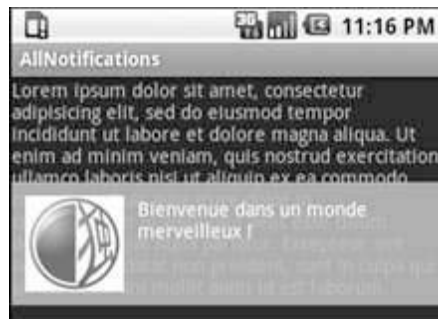


Figure 7.8 — Un toast customisé

7.5.2 Barre de statut

La barre de statut ou barre de notification est le widget horizontal positionné tout en haut de l'écran. Cette zone sert à recevoir des informations un peu à la manière des bandeaux de type « Breaking news » des chaînes d'infos.

Même si la notification commence par s'afficher un bref instant comme ceci :

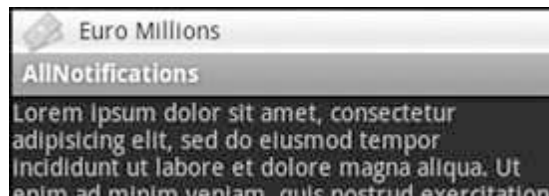


Figure 7.9 — La barre de notification d'Android

pour ensuite disparaître, il est toujours possible ensuite de revenir examiner le contenu de cette notification en tirant le bandeau vers le bas :

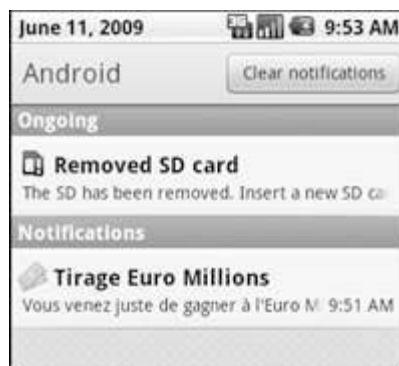


Figure 7.10 — Tous les messages reçus restent accessibles

Le code nécessaire à la publication de la notification précédente est celui-ci :

```
NotificationManager nm = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
int icon = R.drawable.money;
String tickerText="Euro Millions";
long now = System.currentTimeMillis();
Notification notification = new Notification(icon, tickerText, now);
Context context = getApplicationContext();
String title = "Tirage Euro Millions";
String text = "Vous venez juste de gagner à l'Euro Millions !";
Intent notificationIntent = new Intent(this, AllNotificationsActivity.class);
PendingIntent contentIntent = PendingIntent.getActivity(this, 0,
                                                         notificationIntent, 0);
notification.setLatestEventInfo(context, title, text, contentIntent);
nm.notify(WINNING_ID, notification);
```

L'objet *Notification* est créé avec une icône et le texte à afficher dans la barre dans les premiers instants suivants sa réception. Chaque notification est datée : dans l'exemple, l'heure courante est retenue mais il est tout à fait possible d'en choisir une autre.

Ensuite par la méthode *setLatestEventInfo*, le titre et le texte de la notification, visible lorsque la barre est déroulée, sont fixés. Cette méthode accepte également en paramètre un objet de type *PendingIntent*. Pour rappel, un *PendingIntent* encapsule une action à exécuter ultérieurement. En l'occurrence, il s'agit d'un *PendingIntent* créé à partir de la méthode statique *PendingIntent.getActivity*. Cela veut dire que l'action consiste à démarrer l'activité correspond à l'*Intent* notificationIntent. Il s'agit de l'activité *AllNotificationsActivity*. Le *PendingIntent* sera activé lorsque l'utilisateur cliquera sur l'élément de notification dans la barre de statut.

Dans l'exemple, étant donné que l'activité qui est exécutée est *AllNotification-Activity*, il ne se passera rien si ce n'est la fermeture de la barre.

Enfin, la dernière méthode appelée est « notify ». Le premier paramètre est un id de type entier. Cet entier identifie la notification, ce qui permettra de mettre à jour celle-ci par la suite. Par exemple, si l'on devait générer des notifications pour les appels manquants, il pourrait être intéressant de les regrouper par numéro appelant et d'afficher le nombre de fois qu'une même personne a tenté d'appeler.

Enfin, les notifications peuvent également s'accompagner d'un signal sonore, lumineux (LED), ou d'une vibration. Les messages peuvent eux aussi être personnalisés et enrichis d'images et de textes formatés dont la définition se trouve dans un fichier layout XML. Pour ce faire, il faut utiliser la classe *RemoteViews* car le rendu de la *View* ne se fera pas dans le même processus que celui appelant la méthode *notify* du *NotificationManager*.

7.5.3 Les boîtes de dialogue

La dernière alternative pour notifier l'utilisateur est la boîte de dialogue. Il s'agit moins d'une notification mais plutôt d'une sollicitation car une action de la part de l'utilisateur peut être demandée.

Le fonctionnement des boîtes de dialogue sous Android est conforme à celui auquel on peut s'attendre : les fenêtres apparaissent au-dessus de l'activité courante et capturent toutes les actions utilisateurs. Tant que la boîte de dialogue est affichée, l'activité en arrière plan est gelée. La boîte de dialogue est non seulement modale mais étroitement liée à l'activité qui l'a lancée. Boîte de dialogue et activité partagent de nombreuses propriétés comme le menu principal ou le volume sonore.

Dans un souci d'optimisation des ressources et en particulier pour éviter le travail incessant du ramasse-miettes, Android propose un pattern précis pour créer et afficher des boîtes de dialogue.

La méthode `onCreateDialog(int id)` de la classe `Activity` est responsable de la construction de la fenêtre identifiée par l'id passé en argument. Android se chargera de conserver une référence vers chaque dialogue.

La stratégie conseillée est de déclarer autant de constante qu'il y a de type de dialogue à afficher et de procéder à la création des fenêtres à l'intérieur du switch sur l'id :

```
@Override
protected Dialog onCreateDialog(int id) {
    Dialog dialog;
    switch (id) {
        case EXIT_DIALOG:
            AlertDialog.Builder builder = new AlertDialog.Builder(this);
            builder.setMessage("Voulez-vous vraiment arrêter la partie ?");
            builder.setCancelable(false);
            builder.setPositiveButton("Oui",
                new DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog, int id) {
                        AllNotificationsActivity.this.finish();
                    }
                });
            builder.setNegativeButton("Non",
                new DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog, int id) {
                        dialog.cancel();
                    }
                });
            dialog = builder.create();
            break;
        default:
            dialog = null;
    }
    return dialog;
}
```

Ici, une seule constante est définie :

```
private static final int EXIT_DIALOG = 1;
```

Si une autre boîte de dialogue devait être utilisée au sein de l'activité, une autre constante serait déclarée et le switch serait étoffé d'un bloc « case » supplémentaire.

La méthode `onCreateDialog` ne fait qu'instancier les composants graphiques mais ne les affiche pas. Cette méthode ne doit pas être invoquée directement. Elle sera appelée par le système lorsque cela sera nécessaire, c'est-à-dire la première fois que la méthode `showDialog` sera exécutée pour un id donné :

```
■ showDialog(EXIT_DIALOG);
```

L'appel à la méthode `showDialog` (fait depuis le code applicatif cette fois-ci) déclenche l'apparition de la boîte de dialogue :

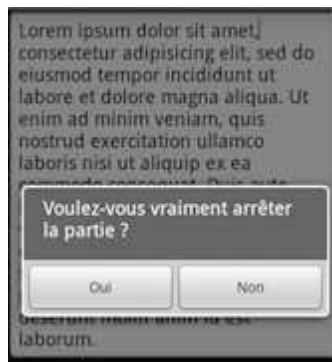


Figure 7.11 — Une boîte de dialogue

À partir du deuxième appel à cette méthode avec le même id, la même instance de dialogue sera réutilisée. La méthode `onCreateDialog` ne sera pas à nouveau exécutée. Grâce à cette gestion des instances par des id de type int, aucune instantiation superflue d'objet n'est faite.

Néanmoins, le maintien de ce cache d'objets peut être assez gênant car il n'est pas rare de vouloir faire varier légèrement la boîte de dialogue d'un affichage à l'autre. Par exemple, on peut imaginer qu'une partie du message soit dynamique.

Heureusement, ce cas-là est prévu. En effet, avant que chaque fenêtre soit rendue visible, la méthode `onPrepareDialog(int id, Dialog dialog)` est appelée avec en paramètre l'objet `Dialog` qui se trouve exactement dans le même état que lors de sa précédente apparition puisqu'il s'agit de la même référence d'objet. Grâce à cette méthode, on a donc la possibilité d'effectuer les ajustements nécessaires.

L'exemple de code ci-dessus illustre la démarche à suivre pour la gestion des boîtes de dialogue avec une fenêtre de type `AlertDialog`. Il s'agit du type générique qui est suffisamment souple (on peut y placer des boutons et d'autres widgets) pour répondre à la plupart des besoins. Mais, il y a en fait plusieurs autres types de fenêtres : les `ProgressDialog` pour rendre compte de l'avancement d'un traitement, les `DatePickerDialog` et `TimePickerDialog` pour sélectionner une date et une heure.

`AlertDialog` ne possède pas de constructeur public. L'instanciation de l'objet se fait par la classe interne `AlertDialog.Builder` qui supervise toutes les étapes de création. Une fois le paramétrage fait, il ne reste alors plus qu'à appeler `create()` pour récupérer la boîte de dialogue.

7.6 2D ET 3D

Android est doté de capacités graphiques 2D et 3D importantes. Au vu du succès des jeux vidéo sur les plateformes mobiles ou des applications nécessitant de réaliser des représentations graphiques complexes et dynamiques comme les logiciels de type « GPS », cela semblait être indispensable.

7.6.1 Graphisme en deux dimensions

Le support des graphismes en deux dimensions est pris en compte par la classe *Drawable* et ses nombreuses dérivées :

- *BitmapDrawable*
- *ClipDrawable*
- *ColorDrawable*
- *GradientDrawable*
- *InsetDrawable*
- *LayerDrawable*
- *NinePatchDrawable*
- *PictureDrawable*
- *RotateDrawable*
- *ScaleDrawable*
- *ShapeDrawable...*

Comme la plupart des composants d'IHM Android, les drawables peuvent s'instancier et s'utiliser dans des fichiers XML ou dans le code Java. Il est même possible d'obtenir un *Drawable* simplement en copiant un fichier image (png, jpeg ou gif) dans le répertoire ressource « *res/drawable* ». Chaque drawable issu d'un fichier image se voit générer un id du type « *R.drawable.le_fichier_image_sans_extension* » pour ce qui est de sa référence dans le code et « *@drawable/le_fichier_image_sans_extension* » pour le XML.

L'ajout de fichiers de ressource image n'est là qu'une manière de créer un drawable. La classe *android.graphics.drawable.Drawable* est en fait une abstraction représentant « quelque chose » qui se dessine à l'écran. Cette classe définit les méthodes de base pour agir sur l'élément devant être dessiné, comme par exemple *setBounds* pour déterminer les dimensions de la fenêtre dans laquelle le composant sera tracé ou *setAlpha* pour fixer le niveau de transparence (de 0 à 255, l'opacité grandissant avec ce nombre). Ensuite, la manière précise dont le drawable sera dessiné ainsi que sa nature véritable dépendent de la sous-classe employée.

Par exemple, avec *LevelListDrawable*, on peut construire un composant graphique un peu plus sophistiqué que la simple image png. Par une définition XML (ici *power_level.xml*), on regroupe plusieurs autres drawables en affectant à chacun un numéro (level) :

```
<level-list xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:maxLevel="0" android:drawable="@drawable/grey" />
  <item android:maxLevel="1" android:drawable="@drawable/yellow" />
  <item android:maxLevel="2" android:drawable="@drawable/green" />
  <item android:maxLevel="3" android:drawable="@drawable/blue" />
  <item android:maxLevel="4" android:drawable="@drawable/red" />
</level-list>
```

Dans le code, le drawable est associé à un widget *ImageView* afin de recevoir les interactions de l'utilisateur :

```
package org.florentgarin.android.graphics;
import android.app.Activity;
import android.content.res.Resources;
import android.graphics.drawable.LevelListDrawable;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.ImageView;
public class GraphicsActivity extends Activity {
    private int m_level=0;
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        Resources res = getResources();
        LevelListDrawable drawable = (LevelListDrawable)
res.getDrawable(R.drawable.power_level);
        final ImageView image = (ImageView)
findViewById(R.id.power_indicator);
        image.setImageDrawable(drawable);
        image.setOnClickListener(new OnClickListener(){
            @Override
            public void onClick(View arg0) {
                image.setImageLevel((++m_level) % 5);
            }
        });
    }
}
```

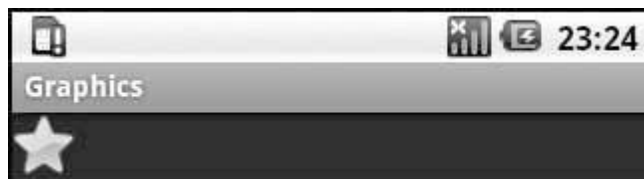


Figure 7.12 — LevelListDrawable à la position 0 (l'étoile est grisée)

À chaque clic, le level est incrémenté, en recommençant à 0 après le niveau 4. Ainsi l'image changera à chaque fois de couleur :

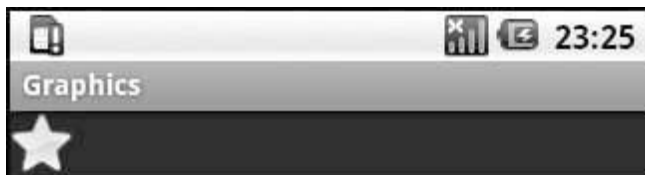


Figure 7.13 — LevelListDrawable à la position 1 (l'étoile est jaune)

Si les nombreux sous types de *Drawable* proposés par Android ne conviennent pas, il sera toujours possible d'utiliser *ShapeDrawable* pour dessiner des primitives graphiques en 2D. *ShapeDrawable* fonctionne en passant à son constructeur un objet de type *Shape* (*RectShape*, *ArcShape*, *OvalShape*, *RoundRectShape*, *PathShape*...) puis en paramétrant le rendu par les méthodes basiques de *Drawable* ou l'objet *Paint* (ex : *drawable.getPaint().setColor(Color.GRAY)*).

Un dernier drawable qui mérite absolument d'être connu est *NinePatchDrawable*. Un *NinePatchDrawable* est une image png, reconnaissable à l'extension .9.png, qui contient des indications qu'Android saura utiliser pour la redimensionner et pour y loger un élément interne.

L'intérêt du *NinePatchDrawable* est de servir d'image de fond à un bouton qui pourra avoir un libellé de taille variable. Contrairement à une image png classique, l'étirement ne se fera pas uniformément dans toutes les directions mais sur des segments préalablement identifiés. Un exemple évident d'usage d'une image nine patch est la création d'un bouton aux bords arrondis. Pour un tel bouton, il n'est pas souhaitable que le redimensionnement touche aux quatre coins : ceux-ci devront demeurer inchangés et cela quelle que soit la longueur du texte à afficher à l'intérieur.

Le terme « nine patch » vient du fait que dans l'exemple plus haut, l'image est divisée en neuf parties :

- les quatre coins ne bougent pas, ils ne sont ni agrandissables ni rétrécissables ;
- les quatre côtés entre les coins sont étirés dans un seul axe ;
- la zone du milieu est redimensionnable dans les deux axes.

Toutefois, le format 9.png est beaucoup plus souple que cela. Il est possible de déterminer les zones pouvant être étendues librement, et il peut donc tout à fait avoir plus de neuf sections comme il peut en avoir moins.

Pour fixer ces zones, il faut tracer sur l'image une bordure d'un pixel d'épaisseur à gauche et en haut pour indiquer les points où l'agrandissement pourra se faire. Android vient avec un utilitaire « *draw9patch* » (qui se trouve dans le répertoire tools) aidant à cette tâche :

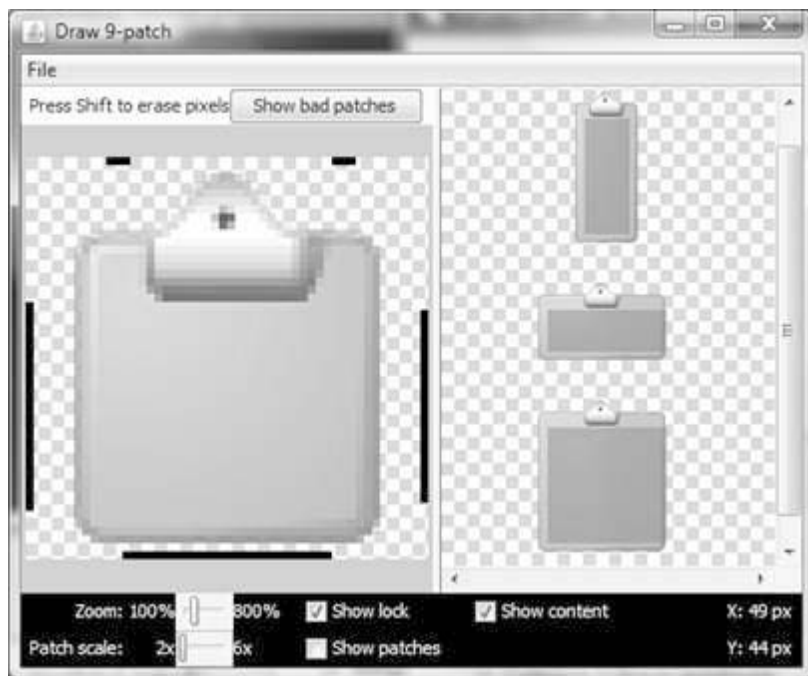


Figure 7.14 — L'utilitaire Draw-9-patch

Après avoir ouvert un fichier png, l'utilitaire permet de définir les bordures et de voir immédiatement le résultat.

Dans l'exemple, on a fixé horizontalement deux zones extensibles de part et d'autre de la pince. Verticalement, il n'y a qu'un seul segment pouvant être agrandi, il ne s'étend pas sur toute la hauteur afin de préserver l'aspect visuel de la pince et du bas de la tablette.

En plus des zones élastiques, les bordures à droite et en bas permettent de délimiter la zone dans laquelle le contenu sera placé. Android agrandira l'image tant que cette zone ne suffira pas pour accueillir le texte du bouton.

7.6.2 Graphisme en trois dimensions

Pour les graphismes 3D, Android propose l'API OpenGL ES 1.0 qui est l'équivalent mobile de l'API OpenGL 1.3. Cette API standard et multilingages, qui n'est pas orientée objet, possède un binding (un portage) Java officiel régi par la JSR 239. Ce binding, conçu pour la plateforme J2ME, est celui qu'Android a également choisi pour son SDK.

Les classes utiles à l'intégration des éléments OpenGL au framework Android sont regroupées sous le package « android.opengl ».

Pour faire cette intégration, le schéma est généralement le suivant :

- on instancie la classe *GLSurfaceView* ;
- par la méthode *setRenderer*, on passe au *GLSurfaceView* l'implémentation de *GLSurfaceView.Renderer* ;
- une implémentation de *GLSurfaceView.Renderer* devra donc être définie, notamment sa méthode *onDrawFrame(GL10 gl)* qui, par l'instance de *GL10* reçue en paramètre, peut effectuer des appels OpenGL.

La documentation d'OpenGL ES n'est pas fournie par Android. La Javadoc du SDK est vide de tout commentaire. L'étude de cette API dépasse le cadre d'Android et de nombreux livres sont exclusivement dédiés à ce sujet.

7.7 APP WIDGETS

Le SDK 1.5 a introduit une nouveauté de taille : l'App Widget. Les App Widgets sont des fragments d'applications qui peuvent être embarqués dans d'autres applications comme par exemple l'écran principal du téléphone. En somme, les App Widgets se basent sur le même paradigme que le dashboard de MacOS ou les gadgets de Vista qui permettent de placer des mini-applications sur le bureau du poste de travail.

La capture d'écran suivante montre le « Home screen » d'Android sur lequel trois App Widgets, une horloge analogique, un lecteur audio et une zone de recherche Google ont été ajoutés :



Figure 7.15 — Trois AppWidgets

La dernière version du SDK (1.5) propose donc un cadre au développement de ce type de composant.

Un App Widget se bâtit sur deux éléments : d'un côté, un *BroadcastReceiver* ou plutôt une sous classe de celui-ci, *AppWidgetProvider*, et de l'autre une hiérarchie de widgets contrôlée avec la classe *RemoteViews*. En effet, l'interface graphique ne sera pas affichée dans le même processus que celui de l'*AppWidgetProvider* qui implémentera la logique de l'application et qui mettra à jour l'IHM. Le processus accueillant la partie visuelle de l'application (dans l'exemple plus haut, il s'agit du Home screen) est dénommé App Widget host.

7.7.1 Définition de l'IHM

L'interface graphique d'un AppWidget se déclare dans un fichier layout XML presque comme n'importe quelle application. « Presque » car comme les widgets seront affichés dans le processus host et mis à jour dans celui de l'*AppWidgetProvider*, l'interface sera basée sur une *RemoteViews*, ce qui impose certaines contraintes.

Les layouts qui pourront être utilisés se limitent aux classes *FrameLayout*, *LinearLayout* et *RelativeLayout*.

Le choix des widgets est lui-même également restreint aux types *AnalogClock*, *Button*, *Chronometer*, *ImageButton*, *ImageView*, *ProgressBar* et *TextView*.

À titre d'exemple, on a choisi de créer un AppWidget qui affichera au hasard une citation qui changera toutes les 10 minutes. Son fichier de layout (*sample_appwidget_layout.xml*) est le suivant :

```
<?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"
    style="@style/WidgetBackground"
>
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/quote"
        android:gravity="center"
        android:layout_marginTop="12dp"
        android:padding="20dp"
    />
</LinearLayout>
```

En examinant ce layout assez classique, on peut tout de même remarquer la définition d'un style. Les styles offrent un moyen pour isoler dans un fichier à part plusieurs directives de formatage qui pourront ensuite être appliquées d'un coup à un ou plusieurs widgets.

Dans le cas présent, le fichier `styles.xml` possède ce contenu

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="WidgetBackground">
        <item name="android:background">@drawable/appwidget_bg_portrait</item>
    </style>
</resources>
```

Ce qui est important ici n'est pas la technique des styles (expliquée dans le chapitre Styles et Thèmes), mais le fait même qu'on applique une image de fond d'écran. Google insiste en effet beaucoup sur la cohérence visuelle entre les `AppWidgets` et édite pour cela des guidelines très précises sur le rendu graphique de ces mini-applications. Parmi ces recommandations, il y en a une au sujet du cadre du widget qui conseille d'adopter le même que ceux des applications de recherche Google ou du lecteur audio. Sur le tutorial Android, il est possible de télécharger plusieurs de ces cadres en fonction des dimensions de l'application.

DP ou DIP (*Density-independent Pixels*) est une unité de mesure indépendante de la densité de pixels qui elle est exprimée en dpi (*dot per inch*), c'est-à-dire en nombre de pixels par pouce (environ 2,54 cm). Le DP est étalonné sur un écran de 160 dpi ce qui veut dire que 1 dp vaudra 1 pixel sur un écran qui a 160 pixels par pouce. Par conséquent, en utilisant l'unité DP au lieu de px (pixel), on s'assure que la taille réelle de l'élément graphique sera approximativement la même quelle que soit la finesse de l'écran.

7.7.2 AppWidgetProvider

Après avoir défini l'interface graphique, on va s'attaquer au cœur de l'application. Pour cela, il faut étendre la classe `AppWidgetProvider` (la classe `SampleAppWidgetProvider` dans l'exemple) qui est elle-même une sous-classe de `BroadcastReceiver`. Le manifeste de l'application déclare ainsi le receiver :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.florentgarin.android.appwidget" android:versionCode="1"
    android:versionName="1.0">
    <application android:icon="@drawable/icon"
        android:label="@string/app_name">
        <receiver android:name="SampleAppWidgetProvider">
            <intent-filter>
                <action android:name="android.appwidget.
                    action.APPWIDGET_UPDATE" />
            </intent-filter>
            <meta-data android:name="android.appwidget.provider"
                android:resource="@xml/sample_appwidget_info" />
        </receiver>
    </application>
    <uses-sdk android:minSdkVersion="3" />
</manifest>
```

La classe `SampleAppWidgetProvider` est donc abonnée à l'événement `android.appwidget.action.APPWIDGET_UPDATE`. Cet événement n'est pas le seul auxquels les *AppWidgetProvider* peuvent s'abonner, néanmoins c'est le seul qui devra être obligatoirement reçu.

L'événement `APPWIDGET_UPDATE` sera émis par le système à intervalle régulier pour déclencher la mise à jour de l'interface (ou des interfaces car l'application peut être ajoutée plusieurs fois) hébergée dans le processus `AppWidget host`.

En plus de la notification d'update, les autres événements sont :

- **ACTION_APPWIDGET_DELETED** – Événement émis lorsqu'une instance d'`AppWidget` est supprimée.
- **ACTION_APPWIDGET_ENABLED** – Émis lorsqu'une instance est ajoutée pour la première fois. L'écoute de cet événement peut permettre la réalisation de tâches d'initialisation.
- **ACTION_APPWIDGET_DISABLED** – Notifie la suppression de la dernière instance d'`AppWidget` pour effectuer d'éventuelles actions de nettoyage.

La déclaration du tag `receiver` du fichier manifeste se termine par la balise méta (obligatoire) dont le nom est « `android.appwidget.provider` » et qui référence le fichier de configuration de l'`AppWidget`.

Dans le cas présent, ce fichier est `sample_appwidget_info.xml` :

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
    android:minWidth="294dp"
    android:minHeight="72dp"
    android:updatePeriodMillis="600000"
    android:initialLayout="@layout/sample_appwidget_layout">
</appwidget-provider>
```

Le fichier de configuration donne les dimensions minimales du widget, la périodicité à laquelle il conviendra d'effectuer les mises à jour ainsi qu'un pointeur vers le layout XML.

Enfin pour finir, voici l'implémentation de la classe `SampleAppWidgetProvider` :

```
package org.florentgarin.android.appwidget;
import android.appwidget.AppWidgetManager;
import android.appwidget.AppWidgetProvider;
import android.content.Context;
import android.widget.RemoteViews;
public class SampleAppWidgetProvider extends AppWidgetProvider {
    private final static String[] QUOTES = {
        "Rien de plus sale que l'amour-propre.",
        "La vie commence là où commence le regard.",
        "L'oisiveté est la mère de la philosophie.",
        "Les idées reçues sont des maladies contagieuses.",
        "Métier d'auteur, métier d'oseur.",
        "La mer enseigne aux marins des rêves que les ports
assassinent.",
        "L'avenir à chaque instant presse le présent d'être un
souvenir.",
```

```

        "Un sac vide tient difficilement debout.",
        "La perfection est une chose insupportable.",
        "L'égalité est la condition de l'échange.",
        "Il y a moins de désordre dans la nature que dans
l'humanité.",

        "Ne pas alourdir ses pensées du poids de ses souliers.",
        "Inventer, c'est penser à côté.",
        "Une erreur originale vaut mieux qu'une vérité banale.",
        "Eclaire demain avec aujourd'hui !" };

@Override
public void onUpdate(Context context,
        AppWidgetManager appWidgetManager,
        int[] appWidgetIds) {
    for (int i = 0; i < appWidgetIds.length; i++) {
        int appWidgetId = appWidgetIds[i];
        RemoteViews views =
            new RemoteViews(context.getPackageName(),
                R.layout.sample_appwidget_layout);
        int random = (int)(Math.random() * QUOTES.length);
        views.setTextViewText(R.id.quote, QUOTES[random]);
        appWidgetManager.updateAppWidget(appWidgetId, views);
    }
}
}

```

La classe *AppWidgetProvider* comme tous les *BroadcastReceiver* possède une méthode public void *onReceive(Context context, Intent intent)* qui est appelée à la réception d'un intent. Dans le cas d'*AppWidgetProvider*, il n'est pas nécessaire de remplacer cette méthode car l'implémentation par défaut est amplement satisfaisante : l'intent reçu est analysé et en fonction de son action, les méthodes *onUpdate*, *onDisabled*, *onEnabled* et *onDeleted* sont invoquées.

Au minimum, la méthode *onUpdate* sera à définir. Cette méthode déclare un paramètre de type *AppWidgetManager* qui permet de communiquer et de mettre à jour les widgets. La méthode a également un autre paramètre, *appWidgetIds*, qui est un tableau de int identifiant la liste des instances dont il faut rafraîchir l'état. Effectivement, il ne faut pas oublier que les *AppWidgets* peuvent être ajoutés plusieurs fois à un même host ou même à plusieurs hosts différents.

L'algorithme de la méthode *onUpdate* est simple : on itère sur les Ids des widgets et pour chacun d'entre eux, on change le *TextView* *R.id.quote* afin d'afficher une nouvelle citation qui aura été tirée au sort. La modification effective se fait avec l'objet *AppWidgetProvider* qui travaille de concert avec la classe *RemoteViews*.

Une fois l'*AppWidget* positionné sur le « bureau » Android, on a le résultat suivant :



Figure 7.16 — Le nouvel AppWidget

La citation apparaît entourée du cadre standard. Le widget s'insère parfaitement dans le design d'ensemble.

7.7.3 Écran de configuration

Le manifeste de l'application de l'exemple ne déclare pas d'activité mais uniquement le receiver `SamplAppWidget` et c'est bien normal car un `AppWidget` n'est pas un programme Android comme les autres qui se lance en démarrant une instance d'`Activity`. Toutefois, on pourrait concevoir une activité particulière destinée au paramétrage du widget, pour, par exemple, sélectionner la ville dont il faudra afficher les informations météorologiques ou la liste des actions dont il conviendra d'afficher le cours.

L'activité de configuration se référence dans le manifeste, elle est sollicitée en réponse à un *Intent* particulier :

```
■ <action android:name="android.appwidget.action.APPWIDGET_CONFIGURE" />
```

En outre, cette activité doit être renseignée dans le fichier de configuration XML `android.appwidget.provider` :

```
■ android:configure="com.example.android.ExampleAppWidgetConfigure"
```

Enfin pour son implémentation, l'activité devra obligatoirement renvoyer un résultat (sous forme d'intention) qui aura dans son bundle extras l'ID de l'`AppWidget` configuré. Cette information sera obtenue depuis l'intention de lancement également dans le bundle extras. Dans les deux cas, l'id sera indexé par la constante `AppWidget-Manager.EXTRA_APPWIDGET_ID`.

7.8 LA NOTION DE TASK

La possibilité d'enchaîner les activités provenant de programmes Android différents est une fonction très puissante qui favorise la mutualisation des modules applicatifs. Cependant, cette pratique a pour conséquence de brouiller un peu la notion d'application. Si, sur le plan purement technique, ces activités peuvent appartenir à des applications Android différentes, déclarées dans des manifestes distincts et s'exécutant dans des processus séparés, pour l'utilisateur, les écrans successifs concourent à réaliser une seule et même tâche et appartiennent donc à la même « application ». C'est par le terme « task » qu'Android désigne cette notion d'application au sens utilisateur.

Un objet task est un empilement d'activités à la manière d'un historique de navigation web. Lorsqu'un programme est démarré en cliquant sur son icône, une tâche est initialisée avec pour premier élément (l'activité racine) l'activité de lancement de l'application. Si cette activité démarre une autre activité, une entrée est ajoutée à la tâche, et si l'utilisateur clique sur le bouton « back », l'activité courante est enlevée et la précédente apparaît.

Si l'utilisateur appuie sur le bouton « Home » et lance un autre programme, une autre tâche est créée ; l'empilement d'activités précédent est néanmoins conservé, il sera à remis au premier plan si l'activité racine venait à être à nouveau exécutée.

Il est toutefois possible d'amender ce fonctionnement par défaut par la déclaration de paramètres sur le tag *Activity* dans le manifeste ou en mentionnant un flag au niveau de l'*Intent* démarrant l'activité.

Par exemple, si la méthode *startActivity* est appelée avec un *Intent* qui a le flag `FLAG_ACTIVITY_NEW_TASK`, la nouvelle activité ne sera pas hébergée sur la même tâche que l'activité qui l'a lancée mais sur la tâche identifiée par l'attribut *affinity* de la nouvelle activité. Si cette tâche n'existe pas encore sur le système, elle est alors créée.

D'autres paramètres peuvent gouverner la politique d'affectation des tâches, par exemple si l'attribut *launchMode* d'une activité a pour valeur « *singleInstance* », cette activité ne pourra pas être instanciée plus d'une fois et sera la seule dans sa tâche.

Le comportement par défaut des tâches convient généralement ; s'il s'avère qu'il faille le redéfinir il faudra lire attentivement la documentation Android pour bien comprendre le résultat qui sortira des combinaisons des divers attributs et des flags des intentions.

7.9 STYLES ET THÈMES

Les fichiers XML de layout, en plus de contenir la description des widgets et leur positionnement, contiennent aussi des informations de formatage des composants comme par exemple la couleur d'un texte, ses marges ou le paramétrage du fond d'écran. Il peut être intéressant d'extraire ces éléments et de les regrouper dans un autre fichier pour d'une part séparer les instructions de formatage, c'est-à-dire la présentation, de la définition du layout proprement dite, la structure, et d'autre part pour pouvoir appliquer à nouveau ce formatage à plusieurs autres instances de composants.

Le principe des styles Android est analogue aux CSS (*Cascading Style Sheets*) qui eux s'appliquent aux tags HTML.

Au paragraphe précédent, le toast personnalisé utilisait un *LinearLayout* qui avait deux attributs de formatage. On pourrait par exemple les réunir dans un style :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="Toast">
        <item name="android:padding">10dp</item>
        <item name="android:background">#DAAA</item>
    </style>
</resources>
```

Cette déclaration de balise style, elle-même englobée dans une balise resources, devra être contenue dans un fichier placé dans « res/values ». La convention veut que ce fichier soit nommé styles.xml et qu'il recense l'ensemble des styles de l'application. Cependant, il n'y a pas d'obligation à cela, il est possible de séparer les styles dans des fichiers différents et de les nommer librement.

Un style comporte un certain nombre de paires clé/valeur portées par le tag item. Ces couples représentent les attributs qui seront affectés aux éléments graphiques. Un style doit déclarer un nom et peut hériter d'un autre style grâce à l'attribut parent. Comme lorsque les attributs sont définis dans les fichiers layout, les valeurs de styles peuvent référencer des ressources avec le caractère « @ ». Ces valeurs peuvent aussi référencer d'autres valeurs du même style, cette fois-ci avec le symbole « ? ».

Ensuite pour effectuer l'affectation, il suffit d'utiliser l'attribut « style » sur le composant dans le fichier de layout :

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/custom_toast"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    style="@style/Toast"
>
```

Le style est donc un moyen efficace d'attribuer en bloc une série de paramètres d'affichage à un widget. Ainsi, en déclarant un style à tous les *TextView*, par exemple, d'une application, on s'assure qu'ils auront tous une apparence homogène. Mais comment s'assurer qu'aucun *TextView* ne sera oublié ou de la cohérence entre tous les types de widgets ? Les thèmes sont une réponse à cette problématique.

Un thème est en fait un style qui s'applique au niveau des activités ou des applications directement dans le fichier manifeste et non plus dans les fichiers de layout.

L'exemple suivant applique le thème « Theme.Dialog » qui est un thème standard inclus sur Android (de nombreux styles prédéfinis existent par défaut) à l'activité *AllNotificationsActivity* :

```
<application android:icon="@drawable/icon" android:label="@string/app_name">
    <activity android:theme="@android:style/Theme.Dialog"
        android:name=".AllNotificationsActivity"
            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>
```

Cela aura pour effet d'assigner ce style à tous les widgets de l'activité. Il est aussi possible d'attribuer un thème à l'application tout entière ce qui équivaut à placer le style sur toutes les activités de l'application.

Par exemple, le thème d'Android « Theme.Dialog » qui donne aux activités un look de boîte de dialogue aboutit au résultat suivant (appliqué à l'activité précédente) :



Figure 7.17 — Thème "Theme.Dialog"

On aperçoit en transparence le rendu du « Home » Android. L'activité apparaît comme s'il s'agissait d'une boîte de dialogue alors que le layout n'a absolument pas été modifié. On voit bien qu'un thème peut considérablement métamorphoser le rendu d'une IHM.

Ce qu'il faut retenir

Sur le plan graphique, Android offre un toolkit particulièrement complet tout à fait à même de concurrencer celui de l'iPhone par exemple. Par son API 3D OpenGL ES, la plateforme est parfaitement adaptée au développement de jeu vidéo.

En frappant fort aux niveaux des capacités visuelles, Google espère probablement rendre Android attractif aux yeux du grand public pour atteindre rapidement une masse critique d'utilisateurs.

8

Interaction avec le matériel

Objectifs

Un smartphone Android est pourvu d'un matériel équipé de fonctionnalités sans équivalent sur un ordinateur de bureau. Ces possibilités nouvelles offertes permettent de concevoir des applications réellement innovantes.

L'écran tactile, le GPS et entre autres l'appareil photo sont entièrement manipulables par l'API Android. Le présent chapitre lève le voile sur ces interfaces de programmation.

8.1 LES FONCTIONS DE TÉLÉPHONIE

8.1.1 Les appels vocaux

Fonction évidemment essentielle au téléphone, la numérotation et le lancement d'appels sont contrôlables par l'API.

L'écran de numérotation est présent sur Android sous la forme d'un composant de type *Activity*. Pour basculer l'affichage sur cette activité et permettre à l'utilisateur de composer un numéro puis ensuite de déclencher l'appel, il suffit de créer un *Intent* avec l'action idoine et d'invoquer la méthode *startActivity* :

```
Intent intent=new Intent(Intent.ACTION_DIAL);  
startActivity(intent);
```

Le choix du numéro reste libre et de la responsabilité de l'utilisateur qui peut même décider de ne pas effectuer d'appel du tout.

Si l'on ne souhaite pas laisser cette possibilité, il faut créer un Intent avec l'action `Intent.ACTION_CALL` :

```
Intent intent=new Intent(Intent.ACTION_CALL);
intent.setData(Uri.parse("tel://0512345678"));
startActivity(intent);
```

L'attribut `data` de l'*Intent* correspond au numéro de téléphone. Dès que l'activité démarre, l'appel est lancé :



Figure 8.1 — Lancement de l'appel

Cette activité, beaucoup plus intrusive que la précédente, exige la permission `android.permission.CALL_PHONE`. Cependant, les numéros d'urgence ne peuvent pas être automatiquement appelés. On peut imaginer que c'est pour éviter les appels intempestifs et non justifiés.

Si l'émission d'appels peut être réalisée par du code, côté réception, l'API propose également quelques fonctionnalités. Plus exactement, il est possible d'enregistrer des listeners pour être informé des changements d'état du téléphone (sonne, en conversation, au repos) :

```
private void registerPhoneListener(){
    TelephonyManager telManager = (TelephonyManager)
getSystemService(Context.TELEPHONY_SERVICE);
    telManager.listen(new PhoneStateListener(){
        public void onCallStateChanged(int state, String incomingNumber){
            super.onCallStateChanged(state, incomingNumber);
            switch(state){
                case TelephonyManager.CALL_STATE_RINGING:
                    Log.i("PlayingWithHardwareActivity",
"Gaston !!!!");
                    break;
            }
        }
    });
}
```

```

    }
    public void onMessageWaitingIndicatorChanged (boolean mwi){
        if(mwi){
            Log.i("PlayingWithHardwareActivity", "Vous avez
reçu un nouveau message.");
        }
    }
}, PhoneStateListener.LISTEN_CALL_STATE |
PhoneStateListener.LISTEN_MESSAGE_WAITING_INDICATOR);
}

```

8.1.2 Réception de SMS

Comme on l'a vu précédemment au chapitre traitant des *BroadcastReceivers*, lorsqu'un SMS est reçu, cette information est véhiculée aux applications par le biais d'un *Intent* qu'il s'agit de capter.

La technique d'écoute des intentions par les *BroadcastReceivers* a déjà été étudiée, on s'attachera plutôt à analyser l'API SMS en elle-même. Pour rappel, voici le code de la méthode *onReceive* du receiver :

```

Bundle extras = intent.getExtras();
SmsMessage[] sms = null;
if (extras != null){
    Object[] pdus = (Object[]) extras.get("pdus");
    sms = new SmsMessage[pdus.length];
    for (int i=0; i<pdus.length; i++){
        sms[i] = SmsMessage.createFromPdu((byte[])pdus[i]);
    }
    String message = "SMS reçu de : ";
    message += sms[0].getOriginatingAddress();
    message += "\n";
    message += sms[0].getMessageBody();
    Toast.makeText(context, message, Toast.LENGTH_LONG).show();
}

```

Les SMS sont contenus dans l'attribut *extras* qui est de type *Bundle*. Un *Bundle* est une sorte de map dont les clés sont un nom (*String*) et les valeurs des objets *Parcelable* (équivalent de *Serializable* dans le monde Android). La valeur intéressante ici est indexée par le nom « *pdus* ». PDU signifie *Protocol Description Unit*, c'est un format utilisé pour encoder les SMS. On passe alors du tableau d'octets représentant les données brutes à l'objet *SmsMessage* par la méthode statique *createFromPdu*. La manipulation du SMS est ensuite aisée grâce aux méthodes dont dispose cette classe. On peut par exemple savoir si le SMS a été envoyé depuis une passerelle mail (*isMail()*).

8.1.3 Envoi de SMS

Pour envoyer des SMS, comme souvent pour les opérations sensibles, l'application devra avoir été autorisée :

```
<uses-permission android:name="android.permission.SEND_SMS">
</uses-permission>
```

Ensuite, la pièce centrale est le *SmsManager* :

```
package org.florentgarin.android.extendedapi;
import android.app.Activity;
import android.os.Bundle;
import android.telephony.gsm.SmsManager;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
public class SendSMSActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.sms);
        Button sendBtn = (Button) findViewById(R.id.send);
        final EditText numberText = (EditText)
findViewById(R.id.number);
        final EditText messageText = (EditText)
findViewById(R.id.message);
        sendBtn.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                String number = numberText.getText().toString();
                String message = messageText.getText().toString();
                SmsManager smsManager = SmsManager.getDefault();
                smsManager.sendTextMessage(number, null, message,
null, null);
            }
        });
    }
}
```

Cet exemple affiche une activité avec une zone de saisie pour le numéro de téléphone du destinataire du SMS, une autre pour le message et un bouton « envoyer ».

Si on ne dispose pas de téléphones physiques, il est quand même possible de tester cette application en lançant deux émulateurs. À la place du numéro de téléphone, il suffit d'entrer le numéro de port de l'émulateur, ce numéro apparaît dans le titre de la fenêtre de l'émulateur.

Le premier émulateur utilise le port 5554 et le second le 5556 :



Figure 8.2 — Échange de SMS entre émulateurs

La méthode `sendTextMessage` du `SmsManager` accepte un paramètre (non utilisé dans l'exemple) intéressant. Il s'agit du `PendingIntent`.

Le `PendingIntent` est une encapsulation d'un `Intent` et de l'action à effectuer avec celui-ci, c'est-à-dire démarrer une activité, un service ou diffuser en direction des `BroadcastReceiver` cet `Intent`. Une instance de `PendingIntent`, `Pending` signifiant en attente, représente une commande qui sera transmise à une autre application : charge à elle de l'exécuter ultérieurement.

Dans l'exemple de l'envoi de SMS, il est possible de passer au `SmsManager` par la méthode `sendTextMessage` un `PendingIntent` qui sera à déclencher au moment où le SMS sera envoyé et un autre au moment où il sera délivré :

```
final PendingIntent sentIntent = PendingIntent.getBroadcast(this, 0,  
    new Intent("SENT"), 0);  
final PendingIntent deliveryIntent = PendingIntent.getBroadcast(this, 0,  
    new Intent("DELIVERED"), 0);
```

On pourrait parallèlement enregistrer un `BroadcastReceiver` capable de recevoir ces deux intentions pour suivre en temps réel l'envoi du SMS.

8.2 GÉOLOCALISATION

L'API de localisation d'Android se divise en deux parties. D'un côté le service de géolocalisation proprement dit dont le rôle est de communiquer avec le matériel, la puce GPS, afin de fournir la position courante. De l'autre côté, l'API Google Maps pour Android dont le but est de représenter visuellement les informations issues de la première partie de l'API.

8.2.1 Service de localisation

Le service de localisation, *android.location.LocationManager*, est un service système dont une référence s'obtient depuis l'activité par la méthode *getSystemService*. L'API de ce service définit une classe abstraite nommée *LocationProvider*. La notion de « fournisseur de localisation » est une abstraction englobant différents moyens concrets de localiser la position du terminal. Parmi ceux-ci, on peut citer le GPS ou la triangulation effectuée à partir des antennes mobiles.

À des fins de test, il est possible d'injecter des données factices à ces providers. Les providers diffèrent entre eux par la précision de leurs informations, la consommation d'énergie qu'ils engendrent ou encore les conditions qui doivent être réunies pour assurer leur fonctionnement (certaines capacités matérielles comme la présence d'une puce GPS, un accès à internet...).

Le programme suivant récupère la position courante de l'appareil et affiche la longitude et latitude correspondant dans deux champs de texte :

```
package org.florentgarin.android.location;
import java.util.List;
import android.app.Activity;
import android.content.Context;
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
import android.os.Bundle;
import android.widget.EditText;
public class LocationSampleActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        final EditText latitude=(EditText) findViewById(R.id.latitude);
        final EditText longitude=(EditText) findViewById(R.id.longitude);
        LocationManager locMngr = (LocationManager)
getSystemService(Context.LOCATION_SERVICE);
        List<String> providers = locMngr.getAllProviders();
        locMngr.requestLocationUpdates(providers.get(0), 0, 0, new
LocationListener(){
            @Override
            public void onLocationChanged(Location loc) {
                latitude.setText(loc.getLatitude()+"");
                longitude.setText(loc.getLongitude()+"");
            }
        });
    }
}
```



```

    }
    @Override
    public void onProviderDisabled(String provider) {
    }
    @Override
    public void onProviderEnabled(String provider) {
    }
    @Override
    public void onStatusChanged(String provider, int
        status,
                                Bundle extras) {
    }
    });
}
}

```

À l'exécution, l'écran suivant s'affiche :

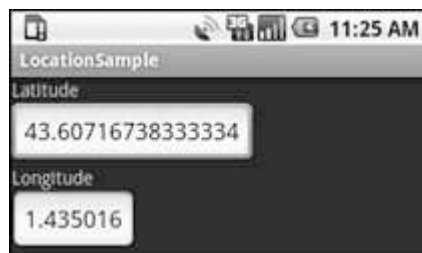


Figure 8.3 — La position courante est affichée

Les coordonnées seront rafraîchies en temps réel car un écouteur *LocationListener* a été enregistré sur le provider.

L'exemple ci-dessus récupère l'ensemble des providers disponibles et choisit de travailler avec le premier de la liste. Il est possible de nommer directement les providers, des constantes existent à cet effet : *LocationManager.GPS_PROVIDER* ou *LocationManager.NETWORK_PROVIDER*. Toutefois, l'intérêt de l'abstraction offerte par les providers est de ne pas avoir à en sélectionner un explicitement.

La méthode *getBestProvider* de la classe *LocationManager* retourne le nom du provider répondant au mieux au critère de sélection (représenté par un objet de type *Criteria*) passé en paramètre. Cet objet possède les attributs suivants, qui sont autant de critères de sélection :

- **speedRequired** - Cet attribut indique si le provider devra pouvoir renseigner la vitesse de déplacement.
- **powerRequirement** - Indique le critère d'exigence sur l'utilisation d'énergie du provider. Cet attribut pourra prendre les valeurs des constantes *NO_REQUIREMENT*, *POWER_LOW*, *POWER_MEDIUM* ou *POWER_HIGH*.
- **costAllowed** - Ce critère booléen détermine si le provider pourra être payant. Pour éviter d'éventuels frais et garantir que seuls les providers gratuits pourront être utilisés, il faut fixer la valeur de l'attribut à faux.

- **bearingRequired** - Le « bearing » est la direction du mouvement. Si cette information devra être lue en positionnant l'attribut à vrai, on exige d'obtenir un provider capable de la fournir.
- **altitudeRequired** - Indique si le provider est tenu de donner l'altitude.
- **accuracy** - Il s'agit de la précision souhaitée pour le provider. Pour une précision maximale la valeur sera fixée à `ACCURACY_FINE` ou pour une précision approximative ce sera `ACCURACY_COARSE`. Par exemple, le GPS est un provider bien plus précis que le provider s'appuyant sur la position des points d'accès Wi-Fi.

Si l'application est lancée depuis l'émulateur, le GPS n'étant pas disponible ni d'ailleurs la puce téléphonique, les champs latitude et longitude resteront désespérément vides à l'écran. Pour y remédier, il faut injecter manuellement des données de localisation pour simuler la présence de vrais providers.

Le *plug-in* Eclipse par la vue Emulator Control, accessible dans la perspective DDMS, permet de saisir ces informations soit en précisant la longitude et la latitude, soit en pointant vers un fichier GPX ou KML.

Le format GPX (GPS eXchange Format) est un format XML ouvert pour l'échange de coordonnées GPS. En plus des simples « waypoints », ce format peut représenter également les « tracks » et les « routes ».

KML (*Keyhole Markup Language*) est un format concurrent destiné au départ au stockage des données géospaciales des logiciels de Google : Google Earth et Google Maps. Néanmoins, d'autres logiciels l'ont aujourd'hui adopté.

Malheureusement, cette fonction du *plug-in* ne semble pas toujours donner entièrement satisfaction ! Il semblerait qu'un bug empêche la transmission des données lorsque le système de l'OS n'est pas en anglais.

Le contournement est heureusement simple : il faut procéder par la ligne de commande, après s'être connecté à l'émulateur par Telnet, il faut entrer l'instruction suivante (exemple avec les coordonnées de Toulouse) :

```
geo fix 1.442951 43.604363
```

8.2.2 API de cartographie

Une fois les informations de localisation obtenues, il faut pouvoir les traiter. Il est fréquent que l'on veuille afficher la position courante sur une carte.

Android ne possède pas officiellement d'API de cartographie. En effet l'API Google Maps, parfaitement opérationnelle sur Android, ne fait pas partie de l'API standard de la plateforme mais de Google API Add-On. Cela veut dire qu'un téléphone labélisé Android ne sera pas forcément pourvu de cette API.

Installation

Le terme « installation » est peut-être excessif mais avant de pouvoir développer avec Google Maps, une phase préparatoire en plusieurs étapes est tout de même nécessaire.

Les propriétés du projet

Pour avoir accès aux classes de l'Add-On, il faut, au niveau des propriétés du projet, clairement notifier que l'on compte importer les classes des API Google :

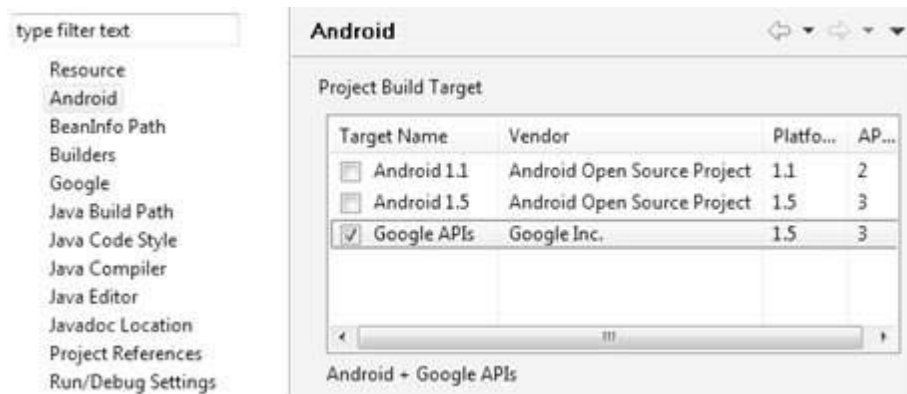


Figure 8.4 — Google APIs intègre Google Maps

Pour ceux qui utilisent ant, pas de souci, il est bien sûr possible de spécifier au build.xml la cible de compilation (*Project Build Target*). La commande Android qui crée un squelette d'application (contenant les fichiers build.xml, build.properties, default.properties, local.properties) accepte le paramètre `--target` afin d'indiquer la cible choisie. Ainsi la commande suivante :

```
android create project --target 3 --path . --package
org.florentgarin.android.location --activity LocationSampleActivity
```

fabriquera un projet dont les fichiers ant de compilation seront configurés correctement pour utiliser l'API Google.

Le projet pourra maintenant faire usage des classes de Google Maps. Cependant, cette API n'est pas une simple librairie Java packagée dans un fichier jar qui pourrait se déployer aisément sur tous les téléphones Android. Au contraire, pour exécuter une application Google Maps, le terminal devra intégrer préalablement l'API.

Construction de l'image AVD

En phase de développement, c'est pareil, l'émulateur devra inclure l'Add-On. Il faudra donc construire une image spéciale de l'émulateur dont la cible « Target » sera bien « Google API ». Depuis Eclipse et le menu « Window/Android AVD Manager », quelques clics suffisent pour cela (figure 8.5).



Figure 8.5 — L'AVD Manager

Pour les réfractaires à Eclipse, grâce au script `tools\android`, il n'est pas plus compliqué de créer des images d'émulateur en ligne de commande qu'avec l'interface graphique :

```
android create avd ---target [plateforme cible] ---name [nom de l'AVD (Android Virtual Device)]
```

Déclaration dans le manifeste

N'étant pas assuré qu'une application Google Maps puisse fonctionner, il est obligatoire d'annoncer dans le manifeste que la librairie est exigée. Ainsi le téléphone pourra en informer l'utilisateur ce qui est toujours préférable à tenter une exécution qui serait vouée à l'échec.

La déclaration se fait entre la balise « application » :

```
<uses-library android:name="com.google.android.maps" />
```

En outre, il ne faut pas oublier d'ajouter la permission donnant l'accès à Internet :

```
<uses-permission android:name="android.permission.INTERNET" />
```

Obtention d'une clé d'utilisation

Google Maps est capable d'afficher la carte du monde entier, sur laquelle il est possible de zoomer pour atteindre un niveau de détail important où les rues, certains bâtiments et le sens de circulation sont identifiés. Google Maps offre même une vue « Satellite » où le plan laisse la place à de véritables photos.

L'ensemble de ces informations, qu'il s'agisse du tracé des routes ou des images, représente un volume considérable. Ces données ne sont pas incluses dans l'API mais téléchargées au besoin depuis les serveurs de Google. À ce titre, la société entend exiger de ses utilisateurs qu'ils approuvent les conditions d'utilisation en requérant la possession d'une clé pour faire fonctionner de Google Maps.

Pour obtenir cette clé, il faut disposer d'un compte Google. Ceux qui n'en ont pas encore peuvent en créer un gratuitement en se rendant à la page : <https://www.google.com/accounts/>

La clé qui sera délivrée sera non seulement rattachée au compte Google mais aussi à un certificat numérique. La clé ne pourra donc être utilisée qu'avec les applications signées à l'aide du certificat. Si l'on change de certificat, par exemple pour packager l'application en vue de sa distribution sur des terminaux physiques, il faudra refaire le processus pour obtenir une nouvelle clé.

Le SDK vient avec un certificat par défaut de debug qui, s'il ne convient pas pour déployer des applications sur les mobiles du commerce, convient amplement pour le travail sur émulateur, y compris pour tester les programmes employant l'API Google Maps.

Pour obtenir la fameuse clé, il faut se munir de l'empreinte MD5 du certificat auquel elle sera liée. Chaque clé est stockée dans un keystore ; pour la clé de debug, celle-ci se trouve dans le fichier `debug.keystore` localisé dans le répertoire « .android » qui se trouve lui-même dans le répertoire home de l'utilisateur.

Sous MS Windows Vista, cela donne :

```
C:\Users\[nom du compte]\.android\debug.keystore
```

Sous Unix :

```
~/.android/debug.keystore
```

L'empreinte MD5 s'obtient en exécutant la commande suivante :

```
keytool -list -alias androiddebugkey -keystore "[chemin du fichier keystore]"  
-storepass android -keypass android
```

L'empreinte apparaît alors sous forme hexadécimale :

```
androiddebugkey, 11 juin 2009, PrivateKeyEntry,  
Empreinte du certificat (MD5) :  
F8:15:40:21:F7:A0:41:BA:BF:65:BC:18:F6:36:61:20
```

Pour obtenir la clé, il faut alors se rendre sur le site de Google à l'adresse : <http://code.google.com/intl/fr/android/maps-api-signup.html> et saisir dans le formulaire l'empreinte que l'on vient de récupérer :

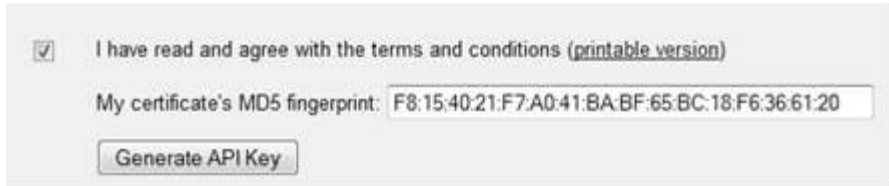


Figure 8.6 — Génération de la clé

Après s'être authentifié avec un compte Google, on obtient enfin la clé :



Figure 8.7 — La clé correspond au certificat

8.2.3 La classe `MapView`

La classe `MapView`, qui hérite de la classe `ViewGroup`, signe qu'on a bien affaire à un widget Android, est un composant représentant des informations géolocalisées sur une carte.

Le widget `MapView` se charge du téléchargement des fragments de carte (les Tiles), de leur mise en cache sur le système de fichiers, des fonctions de zoom et de déplacement de la carte...

Le widget autorise aussi le placement d'éléments visuels, tel que des marqueurs, par-dessus la carte.

La classe `MapView` s'utilise comme n'importe quel widget, soit par une instanciation faite depuis le code soit par une déclaration depuis un fichier de layout :

```
<com.google.android.maps.MapView
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:apiKey="009uXpUR3TpBHdLvFKwrN-625BLK4u5Y5QENmBg"
/>
```

Cependant, le widget présente deux particularités, la première est qu'il faut spécifier une clé d'utilisation, la seconde est que le widget doit être créé dans une activité de type *MapActivity*. Cette activité déclare la méthode abstraite *boolean isRouteDisplayed()*. Il est donc obligatoire d'en fournir une implémentation :

```
@Override
protected boolean isRouteDisplayed() {
    return false;
}
```

Le but de la méthode est simplement de reporter aux services de Google si des informations sur la route (le trafic...) sont affichées à l'écran. Les conditions d'utilisation que l'on a acceptées pour obtenir la clé imposent que cela soit fait honnêtement.

Par défaut, la carte s'affiche sur le pays de l'utilisateur sans possibilité de navigation. Par ajouter un peu d'interactivité et autoriser les déplacements sur la carte, l'attribut *clickable* doit être mis à *true* (*android:clickable="true"*).

Ensuite, pour ajouter un panneau de contrôle pour zoomer sur la carte, il faut faire ceci au niveau du code source :

```
setBuiltInZoomControls(true);
```

MapView possède d'autres paramétrages d'affichage configurables au travers de ses méthodes :

- **setSatellite(boolean)** - Bascule sur la vue satellite.
- **setTraffic(Boolean)** - Place des informations sur le trafic routier.
- **setStreetView(boolean)** - Active le mode "street view". Dans ce mode, il est possible de se ballader dans la ville aux travers d'une succession de photos. Seules les grandes villes sont couvertes.

Toutes ces options sont certes très utiles mais le principal intérêt du *MapView* est de pouvoir centrer précisément la carte sur un lieu géographique ou encore d'y placer des informations issues de l'application.

Le pilotage de la carte se fait par l'objet *MapController*. La méthode *animateTo* prend en paramètre un *GeoPoint* et centre la carte sur celui-ci.

Le *GeoPoint* représente une position géographique où la latitude et la longitude sont exprimées en microdegrés avec des entiers. Les coordonnées en degrés sont donc multipliées par 1 000 000 pour être converties et passées au constructeur de *GeoPoint*.

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    requestWindowFeature(Window.FEATURE_NO_TITLE);
    setContentView(R.layout.map);
    MapView map = (MapView) findViewById(R.id.map);
    map.setBuiltInZoomControls(true);
    MapController mapCtrl = map.getController();
    double lat = 43.604363D;
```

```

        double lng = 1.442951D;
        GeoPoint point = new GeoPoint((int) (lat * 1E6), (int) (lng * 1E6));
        mapCtrl.animateTo(point);
        mapCtrl.setZoom(17);
    }

```

La carte est maintenant centrée sur les coordonnées latitude 43,604363 et longitude 1,442951.

L'ajout d'éléments graphiques sur la carte se fait par la classe *Overlay*. Cette classe est abstraite bien qu'aucune de ses méthodes ne le soit. Pour créer un overlay il faut étendre cette classe et redéfinir une ou plusieurs des méthodes selon le comportement souhaité pour le composant.

Voici ci-dessous un exemple d'overlay qui dessine sur la carte une image, le paramètre *bmp* du constructeur, à l'emplacement géographique spécifié par le paramètre *point* de type *GeoPoint*. Lorsque l'utilisateur cliquera sur l'image, le texte *label* apparaîtra :

```

package org.florentgarin.android.location;
import android.content.Context;
import android.graphics.*;
import android.widget.Toast;
import com.google.android.maps.*;
public class PlacemarkOverlay extends Overlay {
    private Bitmap m_bmp;
    private String m_label;
    private GeoPoint m_point;
    private Context m_context;
    public PlacemarkOverlay(Bitmap bmp, String label, GeoPoint point,
        Context context) {
        m_bmp = bmp;
        m_label = label;
        m_point = point;
        m_context = context;
    }
    @Override
    public void draw(Canvas canvas, MapView mapView, boolean shadow) {
        super.draw(canvas, mapView, shadow);
        Paint paint = new Paint();
        Point screenCoords = new Point();
        mapView.getProjection().toPixels(m_point, screenCoords);
        paint.setStrokeWidth(1);
        paint.setColor(Color.GRAY);
        paint.setAntiAlias(true);
        paint.setStyle(Paint.Style.STROKE);
        canvas.drawBitmap(m_bmp, screenCoords.x, screenCoords.y, paint);
    }
    @Override
    public boolean onTap(GeoPoint p, MapView mapView) {
        Point tapPx = new Point();
        Point markPx = new Point();
        mapView.getProjection().toPixels(p, tapPx);
        mapView.getProjection().toPixels(m_point, markPx);
        if ((tapPx.x > markPx.x - 10) && (tapPx.x < markPx.x + 10)

```



```

        && (tapPx.y > markPx.y - 10) && (tapPx.y <
markPx.y + 10)) {
            Toast.makeText(m_context, m_label,
Toast.LENGTH_LONG).show();
            return true;
        }else
            return false;
    }
}

```

L'objet *Projection*, retourné par la méthode *getProjection()*, est fréquemment utilisé dans les applications Google Maps. Cet objet réalise des transpositions entre les coordonnées en pixels vers les coordonnées géographiques (méthode *fromPixels*) et vice et versa (méthode *toPixels*).

Enfin, il ne reste plus qu'à instancier l'overlay et à l'ajouter à la carte :

```

    Bitmap bmp = BitmapFactory.decodeResource(getResources(),
R.drawable.occitane);
    String label = "La ville rose";
    Overlay mark = new PlacemarkOverlay(bmp,label,point, getBaseContext());
    map.getOverlays().add(mark);

```

On obtient alors :



Figure 8.8 — Un Overlay

En bas de l'écran on peut apercevoir le panneau de contrôle avec les boutons – et + pour faire respectivement un zoom arrière et un zoom avant.

8.3 API RÉSEAU BAS NIVEAU

Android possède bien entendu une API réseau positionnée au niveau de TCP/IP ou HTTP. Toutefois, il peut arriver que l'on ait besoin de descendre les couches OSI. Par exemple, pour une application de streaming vidéo, il peut être intéressant d'adapter la qualité du flux en fonction du réseau utilisé et donc de la bande passante disponible. Ainsi, si l'utilisateur est connecté en edge, pour garantir la bonne fluidité du rendu, il sera nécessaire de diminuer la qualité de l'image par rapport à une connexion Wi-Fi.

Un autre cas où distinguer le réseau physique est important, concerne les applications ouvrant des services mis à disposition par les opérateurs téléphoniques. Ceux-ci voudront probablement réserver l'usage de l'application à leurs clients. Le moyen le plus sûr d'y parvenir est de n'autoriser l'accès que depuis leur propre réseau ; c'est-à-dire de rendre les services inaccessibles depuis une connexion Wi-Fi.

8.3.1 EDGE et 3G

Il n'existe pas véritablement d'API pour contrôler directement les connexions aux réseaux edge ou 3G. Par contre certains services, des « managers » retournés par la méthode `getSystemService`, fournissent des informations sur le réseau et permettent de le superviser.

L'objet `android.net.ConnectivityManager` par ses méthodes `getNetworkPreference()` et `setNetworkPreference(int)` est capable de renvoyer et de spécifier le réseau, `TYPE_MOBILE` ou `TYPE_WIFI`, préférentiel.

La classe `android.telephony.TelephonyManager` définit elle aussi des méthodes utilitaires pour consulter des informations de téléphonie comme le numéro de la carte SIM, le nom de l'opérateur courant, si le téléphone se trouve dans une zone de roaming (en dehors du champ de l'opérateur auquel l'abonnement a été souscrit)...

`TelephonyManager` peut détecter précisément le type de réseau employé contrairement à `ConnectivityManager` qui se contente simplement de dire s'il s'agit du Wi-Fi ou d'un réseau mobile :

```
private void detectNetworkType() {
    TelephonyManager telManager = (TelephonyManager)
getSystemService(Context.TELEPHONY_SERVICE);
    switch (telManager.getNetworkType()) {
        case TelephonyManager.NETWORK_TYPE_UNKNOWN:
            Log.i("PlayingWithHardwareActivity", "type de connexion
inconnu");
            break;
        case TelephonyManager.NETWORK_TYPE_EDGE:
            Log.i("PlayingWithHardwareActivity", "type de connexion edge");
            break;
        case TelephonyManager.NETWORK_TYPE_GPRS:
            Log.i("PlayingWithHardwareActivity", "type de connexion GPRS");
            break;
        case TelephonyManager.NETWORK_TYPE_UMTS:
            Log.i("PlayingWithHardwareActivity", "type de connexion UMTS");
```

```

        break;
    }
}

```

Ces services réclament les permissions `android.permission.ACCESS_NETWORK_STATE` et `android.permission.READ_PHONE_STATE` pour obtenir des informations sur l'état du réseau et `android.permission.CHANGE_NETWORK_STATE` pour en changer.

8.3.2 Wi-Fi

La gestion des connexions Wi-Fi est un point suffisamment important sur un mobile pour mériter un package à part entière ! En effet, « `android.net.wifi` » regroupe les classes permettant le contrôle des aspects Wi-Fi.

Le point d'entrée est la classe *WifiManager*. On obtient une référence vers cet objet grâce à la méthode *getSystemService*. Ensuite, toute une palette de fonctions Wi-Fi devient accessible.

Le *WifiManager* peut donner une multitude de renseignements réseau :

- **getConnectionInfo()** informe sur le nom du SSID (*Service Set Identifier*), c'est-à-dire le nom du point d'accès (en mode infrastructure), l'adresse MAC de la carte Wi-Fi ou encore la vitesse de connexion.
- **getDhcpInfo()** renvoie les informations obtenues depuis le serveur DHCP lorsque la connexion Wi-Fi a été initialisée. Il s'agit des données de configuration classiques qu'un DHCP fournit aux clients : les DNS, la passerelle, l'IP attribué, le masque de sous-réseau et la durée du bail.
- **calculateSignalLevel** donne la qualité du signal.

WifiManager a aussi la capacité de maintenir une liste d'objets de type *WifiConfiguration*. Cette classe encapsule les données de configuration d'un réseau Wi-Fi. La configuration des réseaux auxquels l'utilisateur s'est connecté pourra être conservée ce qui lui évitera entre autres de devoir ressaisir sa clé de sécurité à chaque connexion.

Enfin, le *WifiManager* autorise de scanner à la recherche de points d'accès Wi-Fi (hotspot). La méthode *startScan()* qui déclenche l'opération est asynchrone. Pour être notifié de la fin du traitement, il faut s'abonner à un événement en enregistrant un broadcast receiver :

```

private boolean launchScan() {
    final WifiManager wifi = (WifiManager)
getSystemService(Context.WIFI_SERVICE);
    IntentFilter intentFilter = new IntentFilter();
    intentFilter.addAction(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION);
    registerReceiver(new BroadcastReceiver() {
        public void onReceive(Context context, Intent intent) {
            // l'opération de scan est terminée
            // on récupère les résultats
            List<ScanResult> results=wifi.getScanResults();
            //L'objet ScanResult contient le nom du réseau, son adresse

```

```

        //son schéma d'authentification, le niveau du signal, sa fréquence...
    }
    }, intentFilter);
    boolean initiated = wifi.startScan();
    //le boolean initiated indique
    //si l'opération a pu au moins être lancée
    return initiated;
}

```

8.3.3 Bluetooth

Jusqu'à maintenant, Android ne dispose toujours d'une API Java bluetooth. Néanmoins, elle serait en développement et même aux dires des membres de l'équipe d'Android, sa priorité serait haute.

En revanche, l'absence d'API bluetooth ne signifie pas que les terminaux Android ne supportent pas cette connectivité. Ces derniers, si leurs capacités matérielles le permettent, pourront utiliser des connexions bluetooth. Le support de la norme est toutefois relativement limité, seuls les profils HSP (*Headset Profile*) et HFP (*Hands-Free Profile*) sont pour l'instant fonctionnels.

8.4 L'APPAREIL PHOTO

De nombreux téléphones possèdent aujourd'hui un appareil photo numérique (APN). Android inclut bien entendu un programme servant à prendre des photos.

En plus de ce programme classique, l'API Android offre la possibilité de piloter ce composant. Ce qui permet d'envisager l'APN comme une source d'acquisition de données et d'intégrer finement cette fonctionnalité dans une autre application.

L'objet servant au pilotage de l'appareil photo est *android.hardware.Camera*. Cet objet est en fait un client qui assure la communication avec le service *Camera* qui commande le matériel.

Cet objet ne possède pas de constructeur public : pour obtenir une instance de cet objet, il faut appeler la méthode statique *open*.

Ensuite la prise d'une photo se fait par la méthode *takePicture* :

```

Camera camDevice = Camera.open();
PictureCallback jpegCallback = new PictureCallback() {
    @Override
    public void onPictureTaken(byte[] data, Camera camera) {
        // data est le tableau d'octets
        // de l'image brute non compressée
        Log.i("PlayingWithHardwareActivity",
            "taille de l'image jpeg : " + data.length);
    }
};
camDevice.takePicture(null, null, jpegCallback);
camDevice.release();

```

La prise de la photo se fait de façon asynchrone. Il faut donc enregistrer des callbacks qui seront appelés au fur et à mesure de l'avancement du traitement.

Le premier callback est déclenché juste après que l'image ait été capturée. Les données de l'image ne sont alors pas encore disponibles. Ce callback pourrait être utilisé pour générer un son simulant la mécanique des anciens appareils photos.

Les deux autres callbacks sont de type *Camera.PictureCallback*. Le premier est appelé avec le « raw image data », c'est-à-dire l'image brute non compressée, alors que le dernier réceptionnera l'image au format JPEG. C'est ce callback qui a été utilisé dans l'exemple. L'exécution du code produit les logs suivants :

```
06-13 20:53:56.513: DEBUG/CameraService(542): takePicture
06-13 20:53:56.713: DEBUG/CameraService(542): postRaw
06-13 20:53:56.753: DEBUG/CameraService(542): postJpeg
06-13 20:53:56.833: INFO/PlayingWithHardwareActivity(865): taille de l'image
jpeg : 18474
```

Le callback se contente ici d'afficher la taille en octets de l'image. Dans une véritable application, l'image pourrait être sauvegardée sur le téléphone ou téléchargée sur un serveur distant avec d'autres informations comme les données de géolocalisation ou des données saisies par l'utilisateur.

L'objet *Camera*, en plus de la prise instantanée de photo, a la possibilité d'afficher en continu la prévisualisation de la photo.

Le rendu de la prévisualisation se fait sur un objet *SurfaceView* qui offre une surface sur laquelle il est possible de dessiner. Comme cet objet est de type *View*, il peut être rattaché à une hiérarchie de widgets organisée au sein d'un layout.

L'accès à la surface ne se fait pas directement mais par une instance de *SurfaceHolder* retournée par la méthode *getHolder()*. C'est cet objet qui est passé à l'objet *Camera* par la méthode *setPreviewDisplay(SurfaceHolder holder)* pour lui indiquer où il convient de tracer la prévisualisation. Enfin, pour démarrer la capture en continu, il faut appeler la méthode *startPreview*, la méthode *stopPreview* l'arrêtant.

L'application de démonstration *ApiDemos*, incluse dans le SDK, comporte une activité effectuant le rendu de la prévisualisation à l'écran, sur son content view.

L'émulateur n'ayant pas d'APN, à l'exécution le résultat de la capture est simulée ainsi :

C'est par ce carré qui bouge que l'émulateur représente la vue de la lentille photo.

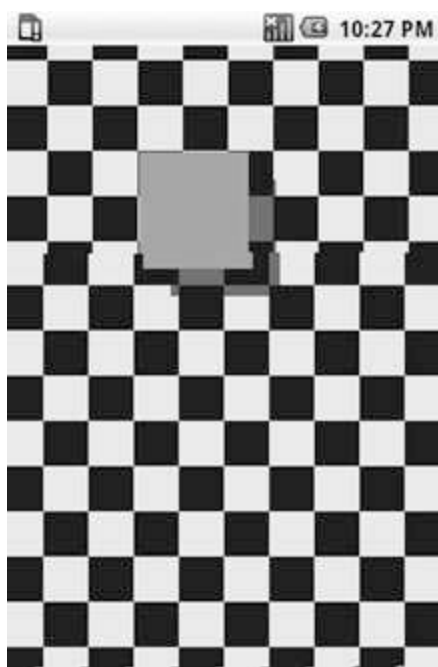


Figure 8.9 — Prévisualisation de l'APN

8.5 API MEDIA

Android est un système d'exploitation doté de capacités multimédia poussées. Une liste d'encoders/decoders audio et vidéo est présente par défaut sur la plateforme.

Tableau 8.1 — Côté decoders

Type	Formats
Image	JPEG, GIF, PNG, BMP
Audio	AAC LC/LTP, HE-AACv1, HE-AACv2, AMR-NB, AMR-WB, MP3, MIDI, Ogg Vorbis, PCM/WAVE
Vidéo	H.263, H.264 AVC, MPEG-4 SP

Tableau 8.2 — Côté encoders, le choix est plus restreint

Type	Formats
Image	JPEG
Audio	AMR-NB
Vidéo	H.263

Le format JPEG, format d'image avec compression destructive, est le format dominant pour l'encodage des photos.

Le format retenu sur Android pour encoder des sources audio est nettement moins connu du grand public, il s'agit du AMR-NB. Ce format est spécialement conçu pour l'encodage de la voix. Il a été adopté par le 3GPP (*3rd Generation Partnership Project*) comme un codec standard et est donc présent sur de nombreux portables.

Le format vidéo H.263 est lui aussi un format répandu dans l'univers du téléphone. Il est certes intrinsèquement moins performant que le format H.264 mais ce dernier a été jugé trop récent et pour maximiser l'interopérabilité avec les autres appareils, l'H.263 a été choisi.

Les constructeurs ont également la possibilité d'ajouter d'autres codecs à leurs mobiles, par exemple les formats propriétaires Microsoft WMA et WMV ne sont pas nativement supportés par Android mais peuvent néanmoins se retrouver sur certains téléphones comme le G1 distribué par T-Mobile.

8.5.1 MediaPlayer

La classe *MediaPlayer* est la classe à partir de laquelle se fait le décodage des fichiers sons et vidéos, c'est-à-dire leur lecture. La lecture des fichiers sonores ne pose aucun problème et deux lignes de code suffisent pour jouer un morceau :

```
MediaPlayer mp = MediaPlayer.create(this, R.raw.song);  
mp.start();
```

Pour cesser la lecture, il faut appeler la méthode *stop*. Lorsque l'on a plus besoin du *MediaPlayer*, il est plus que conseillé d'invoquer la méthode *release* afin de libérer les ressources utilisées par le player interne.

Les données audio peuvent être récupérées depuis une *Uri* ou depuis un fichier ressource propre à l'application placé dans le répertoire « *res/raw* ».

La lecture d'une vidéo est un peu plus délicate. Il faut en effet spécifier au player la zone sur laquelle la vidéo devra être affichée. La logique est la même que celle qui prévaut pour la prévisualisation de l'appareil photo. Il faut se servir de l'objet *SurfaceView* et transmettre son *SurfaceHolder* au *MediaPlayer*.

Toutefois, dans le cas de l'API bas niveau du *MediaPlayer*, *SurfaceView* et *SurfaceHolder* n'est pas nécessaire, il est plus judicieux de se servir du widget *VideoView* qui est plus simple à manipuler.

Le code suivant déclenche la lecture d'une vidéo et place sa zone d'affichage sur le content view de l'activité :

```
/** Called when the activity is first created. */  
@Override  
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    // Hide the window title.  
    requestWindowFeature(Window.FEATURE_NO_TITLE);  
    VideoView vidéo = new VideoView(this);  
    String uriStr = "android.resource://org.florentgarin.android.hardware/"  
        + R.raw.video;
```

```
Uri uri = Uri.parse(uriStr);  
video.setVideoURI(uri);  
setContentView(video);  
video.start();  
}
```

Le composant ne dispose pas de méthode directe pour spécifier comme source vidéo un fichier raw identifié par son id de type `R.raw.resid`. Heureusement, chacune de ces ressources possède une *Uri* de la forme :

```
android.resource://[package_name]/[R.raw.resid]
```

Le *plug-in* Eclipse, par le menu AVD (*Android Virtual Devices*) Manager, permet de créer plusieurs émulateurs et de choisir l'orientation de ceux-ci.

Souvent pour l'affichage des vidéos, le mode paysage est le plus approprié :



Figure 8.10 — Rendu vidéo

La lecture de certaines vidéos peut poser problème. En raison de la très grande variété de codecs, la plupart existant même dans plusieurs versions, il est tout à fait probable qu'une vidéo donnée ne puisse pas être lue par un système Android. Dans ce cas-là, il est indispensable de ré-encoder la vidéo, de nombreux logiciels multimedia proposent ce genre de service avec un format de sortie ciblant spécifiquement Android.

8.5.2 MediaRecorder

L'objet *MediaRecorder* est le miroir du *MediaPlayer*. Son rôle est de capturer une source audio ou vidéo et de la transformer en une série d'octets sauvegardés dans un fichier.

L'enregistrement par le *MediaRecorder* se déroule selon un schéma et une succession d'étapes bien précis.

Pour exemple, le code suivant sauvegarde dans le fichier « *filePath* » les sons capturés par le micro du téléphone pendant une durée « *time* » exprimée en milliseconde :


```
private void recordVoice(String filePath, long time)
    throws Exception {
    MediaRecorder recorder = new MediaRecorder();
    recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
    recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
    recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
    recorder.setOutputFile(filePath);
    recorder.prepare();
    recorder.start();
    Thread.sleep(time);
    recorder.stop();
    recorder.reset();
    recorder.release();
}
```

L'enregistrement démarre avec la méthode *start()*, s'arrête avec la méthode *stop()*. Il faut après avoir effectué le paramétrage de l'enregistrement invoquer la méthode *prepare()* et à la fin de l'utilisation du recorder appeler *release()* pour libérer les ressources sollicitées par l'API. Finalement, rien de bien différent de ce qu'on a vu jusqu'ici.

Durant la phase préparatoire, le format de sortie (output format) et le codeur audio (audio encoder) sont choisis. Pour l'instant, seul l'encoder AMR_NB est disponible. Le format de sortie est le format du conteneur vidéo qui encapsule les données encodées selon l'algorithme du codeur audio sélectionné. Ces conteneurs peuvent, pour certains, regrouper en un même fichier des éléments audio et vidéo encodés différemment. D'autres informations peuvent également s'y retrouver telles que les sous-titres des films par exemple.

L'exemple de code présenté plus haut est assez sommaire : l'enregistrement sonore est sauvegardé dans un fichier sans qu'aucune métadonnée n'y soit rattachée. Or, comme on l'a vu au chapitre sur les *ContentProvider*, ceux-ci sont capables de gérer des données structurées et des données binaires. Le *MediaStore*, qui est un content provider natif sur Android, stocke des données de type média, sons, vidéos, photos...

Grâce à lui, il va être possible de réaliser l'enregistrement micro et de sauvegarder le fichier dans le media store accompagné d'un titre et de sa date de création :

```
ContentValues values = new ContentValues(3);
values.put(MediaStore.Audio.Media.TITLE, "chansons enfants");
values.put(MediaStore.Audio.Media.DATE_ADDED, System.currentTimeMillis());
values.put(MediaStore.Audio.Media.MIME_TYPE, "audio/3gpp");
ContentResolver cr = getContentResolver();
Uri uri =
    getContentResolver().insert(MediaStore.Audio.Media.INTERNAL_CONTENT_URI,
        values);
FileDescriptor fd=cr.openFileDescriptor(uri, "rw").getFileDescriptor();
recorder.setOutputFile(fd);
```

Le reste du programme est ensuite identique.

8.5.3 JET Engine

Android inclut le « JET Engine », un lecteur audio interactif développé par la société SONiVOX. JET se distingue des autres lecteurs audio en ce sens qu'il est capable d'opérer des modulations sonores en temps réel.

L'un des premiers moteurs de ce genre fut iMUSE de la société éditrice de jeux vidéo LucasArts qui l'intégrait dans ses propres productions comme la saga Monkey Island.

On l'a bien compris, cette technologie sert avant tout aux applications ludiques ; l'idée est par exemple d'avoir une musique de fond collant au plus près du déroulement de l'action.

JET se compose de deux éléments : d'un côté l'outil d'autoring, JET Creator avec lequel sera produit un fichier à l'extension .jet, et de l'autre le moteur qui interprétera le contenu du fichier jet avec son API.

JET Creator est un programme implémenté en langage Python avec le toolkit graphique WXWidgets. Avant de pouvoir l'exécuter, il faut bien s'assurer de la disponibilité de ces éléments sur le système.

Le principe de fonctionnement de JET est le suivant : à partir des fichiers sons MIDI importés dans un projet JET Creator, des marqueurs sont ajoutés sur les pistes audio. Ensuite, le fichier jet, produit en sortie, contenant à la fois les métadonnées et les données sonores, deviendra une entrée pour le projet Android ou l'interpréteur JET, piloté par l'API, pourra jouer la bande sonore de manière interactive.

8.6 LE VIBREUR

On désigne par vibreur la fonction matérielle permettant de faire faire au téléphone des mouvements secs, des vibrations. Cette fonction n'est pas nouvelle et n'est pas l'apanage des appareils haut de gamme. Aujourd'hui tous les combinés possèdent cette capacité.

À l'usage, le vibreur s'est révélé être une fonctionnalité très pratique car, grâce à lui, il est possible d'alerter discrètement l'utilisateur. En effet, contrairement à l'alerte sonore, seul le porteur du téléphone ressentira les vibrations et percevra l'information qu'un événement, quel qu'il soit, s'est produit et que s'il souhaite en savoir davantage, il ferait bien de regarder son portable.

Programmatiquement, la classe *android.os.Vibrator* commande cette fonction :

```
Vibrator vibrator = (Vibrator) getSystemService(Context.VIBRATOR_SERVICE);  
vibrator.vibrate(5000);
```

La classe *Vibrator* est récupérée à l'aide de la méthode *getSystemService* qui retourne le service correspondant à la constante fournie en paramètre, ici *Context.VIBRATOR_SERVICE*.

Dans l'exemple, le téléphone vibrera 5 s (5000 milliseconds). La méthode `vibrate` présente une version surchargée qui prend non pas un simple long mais un tableau de long qui contient les durées successives des phases où les vibrations seront actives et non actives. Ce tableau est dénommé `pattern`.

8.7 L'ÉCRAN TACTILE

L'écran tactile est aujourd'hui incontournable sur les téléphones haut de gamme. La suppression ou non du clavier physique fait toujours débat, néanmoins le tactile s'est imposé comme le système de pointage idéal.

Dans le cas d'IHM classique, la gestion des aspects tactiles ne nécessite pas de traitement particulier. L'écoute des événements tactiles se fait au travers d'événements génériques d'un haut niveau d'abstraction comme le listener `View.OnClickListener`.

Par contre, pour une interface graphique plus complexe comme celle d'un jeu vidéo, il est parfois utile de travailler au niveau de l'événement tactile.

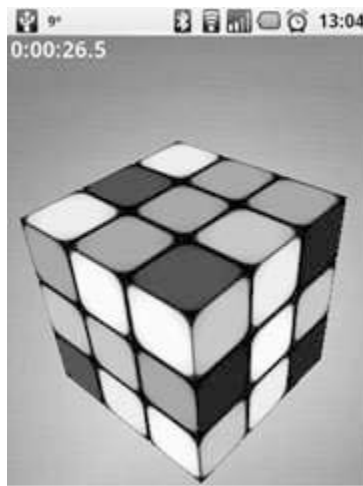


Figure 8.11 — Le cube tourne avec les doigts

Dans un jeu tel quel le célèbre Rubik's Cube, les mouvements de rotation se font très naturellement par le doigt. L'interface utilisateur, le cube en l'occurrence, ne repose pas sur de classiques widgets mais sur une représentation en trois dimensions complexe. Il est donc indispensable de gérer les événements tactiles très précisément en descendant au niveau des coordonnées `x,y`.

Pour cela, on peut soit redéfinir la méthode `public boolean dispatchTouchEvent (MotionEvent ev)` de l'activité, soit la méthode `public boolean onTouchEvent (MotionEvent event)` qui existe également sur l'activité.

La première est invoquée en amont, avant de transmettre l'événement aux objets `View` de l'activité. La seconde ne le sera que si l'événement `MotionEvent` mis en

paramètre n'est pris en charge par aucun des éventuels widgets de l'activité. Le contrat des deux méthodes prévoit de retourner vrai si l'événement a été consommé (géré) et faux dans le cas contraire. La majorité du temps, il est préférable de redéfinir la méthode *onTouchEvent*, celle-ci étant moins exclusive car seuls les événements non traités déclenchent l'appel à la méthode.

Les informations relatives à l'événement reçu sont encapsulées dans un objet de la classe *MotionEvent*. On y retrouve bien entendu les coordonnées de l'événement mais aussi le degré de la pression qui a été exercée, sa durée, les « modifier keys » qui ont été tapés en même temps que l'événement tactile se produisait (touche ALT, CTRL ou SYM, permettant la programmation de combinaisons), la nature même de l'action, c'est-à-dire s'il s'agit d'une pression du doigt proprement dite ou du relâchement d'une pression précédente ou encore d'un mouvement (gesture)...

Le package « android.test » contient la classe utilitaire *TouchUtils*. Cette classe déclare de nombreuses méthodes servant à simuler des événements tactiles allant du simple tapotage de l'écran au déplacement du doigt dessus en passant par le défilement de l'écran.

Cette classe est destinée à être employée dans le cadre de tests réalisés grâce au framework JUnit, dont une adaptation à Android est fournie dans le SDK.

8.8 L'ACCÉLÉROMÈTRE

L'accéléromètre est le composant matériel qui détecte les mouvements subis par le téléphone. Cette capacité que les téléphones modernes sont de plus en plus nombreux à posséder est une façon inédite que l'utilisateur a de communiquer avec le terminal. Très prisé dans les jeux vidéo où piloter une voiture ou un avion en inclinant le téléphone peut sembler tout indiqué, cette fonctionnalité n'en est pas moins intéressante dans le cadre d'applications non ludiques.

Comme souvent, lorsqu'il s'agit d'interagir avec le matériel, il faut obtenir l'objet « manager » depuis la méthode *getSystemService(String name)* de l'activité :

```
SensorManager sensor = (SensorManager)
getSystemService(Context.SENSOR_SERVICE);
sensor.registerListener(new SensorListener() {
    @Override
    public void onAccuracyChanged(int sensor, int accuracy) {
        // changement du niveau de précision
    }
    @Override
    public void onSensorChanged(int sensor, float[] values) {
        // information (values) provenant du sensor identifié
        // par le paramètre de même nom
        switch (sensor) {
            case SensorManager.SENSOR_ACCELEROMETER:
                // ...
            case SensorManager.SENSOR_MAGNETIC_FIELD:
                // ...
            case SensorManager.SENSOR_ORIENTATION:
```

```
        // ...  
        case SensorManager.SENSOR_DELAY_FASTEST:  
            // ...  
        }  
    }, SensorManager.SENSOR_ACCELEROMETER  
    | SensorManager.SENSOR_MAGNETIC_FIELD  
    | SensorManager.SENSOR_ORIENTATION,  
    SensorManager.SENSOR_DELAY_FASTEST);
```

Ensuite, il convient de s'abonner à un ou plusieurs capteurs en fournissant un masque de bits.

Attention toutefois à bien sélectionner les capteurs auxquels s'enregistrer. Les informations reçues peuvent être nombreuses, il est souhaitable de bien cibler les capteurs et de bien choisir la précision (accuracy).

Ce qu'il faut retenir

Le SDK Android couvre relativement bien la partie matérielle des téléphones de nouvelle génération. Toutefois, cette API n'est pas encore aujourd'hui (à la version 1.5 du SDK) entièrement complète. En effet, le Bluetooth est complètement absent de l'API.

Au niveau de la géolocalisation, pour représenter visuellement des données géographiques, Android bénéficie de ce qui se fait de mieux : Google Maps.

La version mobile du programme de Google n'a rien à envier à celle accessible depuis le web sur un navigateur. Tout y est, la vue du trafic, la vue satellite, même Street View est disponible.

Par contre, il ne faut pas perdre d'esprit que l'API Google Maps est une extension et ne fait pas partie du SDK officiel d'Android. Cela veut dire qu'un programme s'appuyant sur Google Maps ne pourra pas être assuré de fonctionner sur n'importe quel téléphone Android.

En outre, un enregistrement en vue de l'obtention d'une clé (gratuite) d'utilisation est nécessaire.

9

Le réseau

Objectifs

Il n'échappe à personne qu'Android est un système communicant. Les composants, *Activity*, *Service*, *ContentProvider* et *BroadcastReceiver*, ont tous des interfaces de communication.

Toutefois, cette communication reste bornée aux applications internes des téléphones.

Ce chapitre est dédié à la communication avec le monde extérieur, ce pourquoi un téléphone est finalement fait.

9.1 INTÉGRATION WEB AVEC WEBKIT

Comme on le sait, WebKit est le moteur de rendu HTML/JavaScript d'Android. Ce moteur se retrouve bien sûr dans le navigateur web mais il est mis à disposition de tous les développeurs dans l'API par le widget *WebView*.

Avec le composant *WebView*, il est possible d'encapsuler au sein d'une interface graphique classique une fenêtre web affichant le contenu dont l'URL aura été spécifiée.

Le *WebView* est également capable de réaliser le rendu d'éléments (HTML, images...) stockés localement mais le sujet du chapitre n'est pas là. On s'attache ici à aborder le widget sous l'angle de sa capacité à communiquer.

9.1.1 Approche mixte : web et native

Intégrer un *WebView* dans une interface native est une approche très intéressante à plusieurs titres.

Premièrement, cela permet éventuellement de recycler une partie d'une application web existante. Ensuite, même dans le cas d'un nouveau développement, il peut y avoir un intérêt à s'appuyer sur ce composant pour construire un pan de l'application qui, cette fois-ci, pourra être réutilisé en dehors d'Android, en particulier sur d'autres téléphones, notamment ceux bénéficiant du même moteur webkit, et ils sont nombreux. En outre, les développeurs maîtrisant les technologies web sont logiquement moins rares que ceux ayant une expérience solide sur une technologie aussi récente qu'Android.

On retombe en somme dans le débat application native contre application web. Cependant, grâce au *WebView*, il est possible de tirer profit des deux approches en les mixant au sein de la même application.

Ainsi, les composants natifs de la plateforme se chargeraient de l'accès aux données locales tel que le carnet d'adresses ou la bibliothèque multimédia, le dialogue avec le matériel (envois de SMS...) pendant que des éléments web intégrés par *WebView* prendraient à leur compte l'affichage et la mise à jour des données hébergées sur un serveur distant.

Deux mondes pas si étanches que ça

L'approche mixte réunit deux univers très différents qui sont, d'un côté un morceau d'application fait de JavaScript, CSS, HTML interrogeant par HTTP un serveur, et de l'autre côté un deuxième bout d'application constitué de composants purement Android : widgets, services, content providers, broadcast receivers...

Néanmoins, ces deux galaxies peuvent communiquer entre elles !

En effet, la classe *WebView* par sa méthode *addJavascriptInterface* permet d'exposer un objet Java depuis le contenu web en JavaScript. L'approche mixte est donc une option d'architecture véritablement fiable et industrielle.

9.2 CONNEXION DIRECTE AU SERVEUR

L'approche mixte précédemment évoquée est intéressante certes mais on peut vouloir mettre en place un dialogue plus direct avec le serveur que simplement ouvrir une fenêtre web à l'intérieur d'une application Android.

9.2.1 Les bases de données

La présence des packages « *java.sql* » et « *javax.sql* » peut laisser entendre qu'il est aisé de se connecter à une base de données distante directement depuis une application Android à l'aide de l'API JDBC. Hélas il n'en est rien.

Par exemple, le code suivant ne fonctionne pas :

```
try {
    Class.forName("com.mysql.jdbc.Driver");
    Connection conn = DriverManager.getConnection(
        "jdbc:mysql://monserveur/mabd","root","root");
    Statement stm = conn.createStatement();
    ResultSet rst = stm.executeQuery("SELECT * FROM MATABLE");
    while(rst.next()){
        String name= rst.getString("MACOLONNE");
        Log.i(DatabaseActivity.class.getName(), "macolonne : " + name);
    }
    rst.close();
    stm.close();
    conn.close();
} catch (Exception e) {
    Log.e(DatabaseActivity.class.getName(), e.getMessage());
}
```

C'est pourtant une utilisation classique de l'API JDBC où, dans un premier temps, on charge le driver, en l'occurrence celui de MySQL, puis on ouvre une connexion pour ensuite exécuter une requête de sélection.

Malheureusement, à l'exécution, les logs d'erreurs ci-dessous sont produits :

```
05-31 22:36:19.657: DEBUG/dalvikvm(209): GC freed 3778 objects / 242456 bytes
in 178ms
05-31 22:36:20.059: WARN/dalvikvm(209): VFY: unable to find class referenced
in signature (Ljavax/naming/Reference;)
05-31 22:36:20.059: WARN/dalvikvm(209): VFY: unable to resolve virtual method
5738: Ljavax/naming/Reference;.get (Ljava/lang/String;)Ljavax/naming/RefAddr;
05-31 22:36:20.059: WARN/dalvikvm(209): VFY: rejecting opcode 0x6e at 0x0004
05-31 22:36:20.059: WARN/dalvikvm(209): VFY: rejected
Lcom/mysql/jdbc/ConnectionPropertiesImpl$ConnectionProperty;.initializeFrom
(Ljavax/naming/Reference;)V
05-31 22:36:20.067: WARN/dalvikvm(209): Verifier rejected class
Lcom/mysql/jdbc/ConnectionPropertiesImpl$ConnectionProperty;
05-31 22:36:20.067: WARN/dalvikvm(209): Unable to match class for part:
'Lcom/mysql/jdbc/ConnectionPropertiesImpl$BooleanConnectionProperty;'
05-31 22:36:20.067: WARN/dalvikvm(209): Exception Ljava/lang/RuntimeException;
thrown during Lcom/mysql/jdbc/ConnectionPropertiesImpl;.<clinit>
05-31 22:36:20.067: DEBUG/AndroidRuntime(209): Shutting down VM
```

Le responsable de ce fiasco ? Le driver JDBC !

En effet, ce driver, fourni par le constructeur de la base de données, se présente sous la forme d'une librairie Java, un fichier jar, contenant les fichiers compilés « .class » implémentant l'API JDBC. Cette librairie encapsule le protocole d'échange spécifique au serveur de base de données.

Pour pouvoir dialoguer avec une base de données quelle qu'elle soit, il convient donc de positionner ce driver dans le classpath de la machine virtuelle. Dans l'exemple, il s'agissait du fichier `mysql-connector-java-5.1.5-bin.jar` qui a bien été référencé en tant que librairie au niveau du projet Eclipse.

Le souci vient du fait qu'Android n'est pas une machine virtuelle JVM et n'exécute donc pas le bytecode des fichiers « .class ». Comme on l'a vu lors de la présentation du framework, Android transforme les .class dans un format qui lui est propre, le format dex. C'est dans ce format que l'application sera déployée puis exécutée sur le téléphone.

La présence d'une librairie dans un projet Android n'est pas un problème en soit. Lors du packaging de l'application, le *plug-in* Eclipse se chargera de convertir les classes du jar dans le format dex.

Ce qui pose problème est que Dalvik, la machine virtuelle d'Android, ne dispose pas de l'intégralité de l'API Java SE (*Standard Edition*). Ainsi, il suffit qu'une librairie tierce s'appuie sur une classe ne faisant pas partie du sous-ensemble de l'API Java standard porté sur Android pour que cette librairie ne soit pas en mesure de fonctionner.

Un mal pour un bien ?

Opérer une connexion directe à la base de données depuis le téléphone Android n'est pas conseillé d'un point de vue architecture logicielle.

Il est en effet préférable de passer par un middleware. Cette couche serveur intermédiaire sera la seule habilitée à se connecter à la base de données, ce qui est plus sécurisé que d'ouvrir l'accès à la base aux connexions externes. En outre le schéma de celle-ci n'est alors pas exposé aux clients.

Ces derniers ne sont couplés au serveur qu'au travers de l'interface de communication haut niveau qui est beaucoup plus fonctionnelle qu'un schéma relationnel.

En conclusion, si dans le cadre d'un développement d'application métier, cette impossibilité de se connecter à la base de données n'est pas vraiment une limitation gênante, malgré tout, cette contrainte signifie qu'il ne sera pas possible de développer d'outil tel que Toad ou un client jdbc universel comme Squirrel pourtant très utiles pour administrer une base.

9.2.2 Quels protocoles réseaux utiliser ?

L'accès direct à une base de données étant interdit, il faut passer par un middleware. Soit, mais quel protocole utiliser pour communiquer avec ce middleware ?

RMI (JRMP) ? Corba (IIOP) ?

Les packages « java.net » et « javax.net » sont bel et bien présents dans le SDK Android mais pas de trace des packages « java.rmi.* », « javax.rmi.* » ni « org.omg.CORBA.* ». Donc nativement, Android ne supporte ni RMI ni Corba.

Alors on peut toujours s'amuser à essayer d'autres implémentations du standard Corba comme JacORB (<http://www.jacorb.org/>) et éventuellement adapter les classes qui poseraient problèmes (la licence, LGPL, très permissive de JacORB l'autoriserait), mais ce serait sans doute aller à l'encontre du positionnement d'Android.

Android est une plateforme Internet : il a donc été conçu autour des standards du web. Par conséquent, le protocole de transport de prédilection du système est l'http.

9.2.3 Les web services

Le service web est l'évolution des anciens systèmes RPC comme RMI, CORBA ou DCOM. Les services web sont nés du constat du relatif échec des technologies précédemment citées qui n'ont jamais véritablement trouvé leur place en dehors du réseau local de l'entreprise.

Au milieu des années quatre-vingt-dix, nombreux sont ceux qui pensaient que très rapidement, grâce à Corba, on trouverait une multitude de services accessibles sur le réseau des réseaux Internet. L'entreprise Netscape bâtit d'ailleurs en grande partie sa stratégie de développement sur cette espérance.

Malheureusement, le résultat n'a jamais été au rendez-vous. La raison principale étant que Corba n'a pas été pensé pour fonctionner dans un environnement étendu et hétérogène comme Internet. Les web services entendent donc corriger ce défaut de conception initial en basant leur architecture sur les technologies fondatrices du web.

Notamment, le protocole de transport devra être en mesure de transmettre les requêtes même à travers un environnement dégradé avec la présence de pare-feu et de proxies.

Cependant, tout n'est pas rose, les spécifications des web services, à force de s'étoffer d'année en année, ont fini par atteindre un niveau de complexité tel que l'interopérabilité promise est aujourd'hui menacée. Si bien qu'aujourd'hui, plusieurs écoles s'affrontent ; d'un côté ceux qui souhaitent tout spécifier et ne laisser aucune place à l'improvisation, et de l'autre les partisans de la simplicité.

9.2.4 SOAP

SOAP signifiait *Simple Object Access Protocol*. Il faut bien dire signifiait car depuis la version 1.2 du protocole, le terme SOAP n'est plus considéré comme un acronyme. Les mauvaises langues diront que de toute façon, l'adjectif « simple » n'était plus de mise.

Le protocole est basé sur le XML mais est indépendant de tout protocole de transport. On peut donc véhiculer un message SOAP aussi bien avec SMTP qu'avec HTTP, même si dans la pratique, c'est ce dernier qui est le plus souvent employé.

SOAP s'accompagne d'une myriade d'autres spécifications :

- **WSDL** (*Web Services Description Language*) – Il s'agit de la description du service dans un langage lui aussi de type XML. Un classique, les services Corba sont décrits en IDL, les services Android en AIDL et les services SOAP grâce au langage WSDL.
- **WS-*** – Les spécifications autour de SOAP sont si nombreuses que c'est ainsi qu'on les désigne ! Parmi elles, on peut en retenir quelques-uns comme :
 - **WS Policy** permet au service de déclarer sa politique, des exigences en matière de sécurité, qualité de service...

- **WS Addressing** sert à mentionner les informations d'adressage des messages indépendamment du protocole de transport, sans s'appuyer par exemple sur les caractéristiques d'HTTP.
- **WS Security** norme relative à la sécurité au sens large : intégrité, confidentialité des messages...

• ...

Pour revenir au sujet Android, on voit bien là que, sans vouloir prendre part au débat des pros et des anti-SOAP, cette avalanche de normes a de fortes chances de poser problème pour un terminal Android aux capacités restreintes.

Android ne possède donc pas nativement de pile SOAP, les concepteurs du système ayant probablement estimé qu'un protocole plus léger était souhaitable. Néanmoins, tout n'est pas perdu : le projet open source KSOAP2 apporte tant bien que mal à Android cet élément manquant.

Selon sa propre définition, KSOAP2 est un projet visant à fournir une implémentation de SOAP aux environnements contraints tel que les applets et les mobiles JavaME. KSOAP2 n'est donc pas un projet dédié à Android, il a d'ailleurs vu le jour avant que ce dernier n'existe.

Pour pouvoir utiliser KSOAP2 sur Android, quelques acrobaties s'imposent donc malheureusement. Une fois récupéré le jar de la librairie, il convient en plus de développer une implémentation compatible Android de la classe *org.ksoap2.transport.Transport* et de l'interface *org.ksoap2.transport.ServiceConnection*.

Sur les forums et sur le Google Groupe consacré à Android, on trouve aisément ces deux classes. Cependant attention, les classes téléchargeables sur Internet ne sont pas compatibles avec les SDK 1.1 d'Android et supérieurs car elle s'appuie sur la librairie Apache HttpClient 3.1. Or dorénavant, Android intègre cette librairie en version 4.

Il faut donc sur la base de cette version effectuer un petit travail de migration ; l'API HttpClient version4 ayant à présent pour classe centrale *DefaultHttpClient*.

Pour pouvoir éventuellement s'authentifier selon la méthode authentification Basic, il faudra amender légèrement la méthode call de la classe *AndroidHttpTransport* (toujours sur la base de la version trouvable sur Internet) pour ajouter dans le header http les credentials :

```
...
if (login != null && password != null) {
    String auth = new String(Base64.encode((login + ':' +
password).getBytes())).trim();
    connection.setRequestProperty("Authorization", "Basic " + auth);
}
...
```

Enfin, pour envoyer effectivement la requête SOAP :

```
package org.florentgarin.android.soapclient;
import org.ksoap2.SoapEnvelope;
import org.ksoap2.serialization.SoapObject;
import org.ksoap2.serialization.SoapSerializationEnvelope;
import org.ksoap2.transport.AndroidHttpTransport;
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
public class SOAPClientActivity extends Activity {
    private static final String SOAP_ACTION = "A définir";
    private static final String METHOD_NAME = "A définir";
    private static final String NAMESPACE = "A définir";
    private static final String URL = "A définir";
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        SoapObject request = new SoapObject(NAMESPACE, METHOD_NAME);
        SoapSerializationEnvelope envelope = new
        SoapSerializationEnvelope(SoapEnvelope.VERSION1);
        envelope.env=SoapEnvelope.ENV;
        envelope.setOutputSoapObject(request);
        AndroidHttpTransport androidHttpTransport = new AndroidHttpTransport(URL,
        "login","password");
        androidHttpTransport.debug=true;
        try {
            androidHttpTransport.call(SOAP_ACTION, envelope);
            Object result = envelope.getResponse();
            Log.i("SOAPClientActivity",result.toString());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Et voilà, il ne reste plus qu'à prier pour que le service en face utilise la même version de SOAP, le même encodage, le même style et qu'il aura soigneusement évité d'utiliser MTOM ou une des normes WS-* qui représentent autant de risques potentiels d'incompatibilité. Enfin, ces problématiques dépassent le cadre d'Android...

Tout n'est toutefois pas négatif, interroger un web service SOAP depuis Android est réellement possible, ce qui est rassurant si le service en question existe déjà. Par contre, dans le cas d'un développement nouveau où le service sera conçu exclusivement pour le client Android, il est plus prudent de ne pas retenir SOAP comme le protocole de communication.

Une fois de plus, il est bon de rappeler qu'Android ne possède pas de pile SOAP, que le projet KSOAP2 n'a pas été créé pour Android et que ce projet n'a pas une activité importante. La prudence est donc de mise.

9.2.5 POX (Plain Old XML)

Face aux critiques sur la complexité de SOAP, certains prônent le retour à la simplicité : continuer certes à représenter les données sous une forme XML mais sans s'embarasser des multiples couches de spécifications. Ici, il n'est nullement question de contraintes sur l'en-tête HTTP auxquelles s'ajouteraient des contraintes sur un format d'encapsulation comme l'enveloppe SOAP.

Seul le message utile est transmis, diminuant ainsi la verbosité du protocole et la masse de traitements à opérer mais surtout, grâce à une simplicité accrue, l'interopérabilité s'en trouve renforcée.

Pour manipuler les messages XML, Android dispose des traditionnels parsers DOM et SAX, mais également de l'API XMLPull v1.

Pour envoyer puis recevoir ces messages, il faut toujours utiliser le module HttpClient d'Apache. Ce composant est une implémentation complète du protocole HTTP côté client. Il permet d'émettre des requêtes selon les différentes méthodes HTTP GET, POST, PUT, DELETE, ce qui rend le composant parfaitement apte à dialoguer avec un service REST.

L'API gère aussi les cookies, les proxies, le SSL, l'authentification de type Basic, Digest et même NTLM (protocole d'authentification propriétaire de Microsoft).

L'emploi de simples messages POX permet donc de simplifier le format d'échange au point qu'il devient possible de « fabriquer » la trame HTTP à la main. Néanmoins, ce retour aux fondamentaux pourrait paraître quelque peu rustique à tous ceux habitués aux communications RPC, comme RMI, qui se font de manière transparente au travers d'objets proxy.

Toutefois, spécifiquement au sujet d'Android, ceux-là n'auront pas à regretter SOAP pour cette raison car la librairie KSOAP2 ne propose de toute façon ni la génération dynamique ni la génération statique du stub client à partir du WSDL.

Google et SOAP

Google ne semble pas miser beaucoup sur SOAP.

En 2006, Google avait déprécié son API SOAP Search en cessant de délivrer des clés d'utilisation. Google recommandait alors d'utiliser plutôt son API AJAX (JavaScript).

Bien que ce choix ne soit sans doute pas uniquement lié à des considérations techniques, en privilégiant le JavaScript Google qui favorise l'utilisation directe du service (de l'utilisateur final vers ses serveurs) et en limitant les possibilités de sauvegarder les résultats des recherches, il est quand même un signe de la défiance que Google a vis-à-vis de SOAP quant à sa capacité à être facilement utilisé par un large public employant des technologies hétéroclites.

9.2.6 JSON (JavaScript Object Notation)

JSON est un format de données textuel issu du JavaScript (ECMAScript pour plus exact) où il était employé comme une syntaxe pour décrire les valeurs des instances d'objets.

Contrairement au XML qui peut représenter des données orientées document, JSON se focalise sur la description d'objets.

Un autre avantage reconnu de JSON par rapport à XML est qu'il est nettement moins verbeux que ce dernier.

Quoi qu'il en soit, JSON ou POX, la philosophie des services web exposant une interface d'échange reposant sur l'un ou l'autre des formats est la même : il s'agit d'envoyer et de recevoir des informations dans un format facilement manipulable par le protocole de transport HTTP.

L'exemple ci-dessous envoie une requête au moteur de recherche Yahoo! grâce à son API Web Search :

```
package org.florentgarin.android.json;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.DefaultHttpClient;
import org.json.JSONArray;
import org.json.JSONObject;
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
public class JSONClientActivity extends Activity {
    private final static String BASE_URL =
"http://search.yahooapis.com/WebSearchService/V1/webSearch?appid=YahooDemo";
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        search("federer", 3);
    }
    private void search(String textSearch, int results) {
        String url = BASE_URL + "&query=" + textSearch + "&results=" +
results
        + "&output=json";
        HttpClient httpClient = new DefaultHttpClient();
        HttpGet httpget = new HttpGet(url);
        HttpResponse response;
        try {
            response = httpClient.execute(httpget);
            Log.i("JSONClientActivity", "StatusLine : "
+ response.getStatusLine().toString());
            HttpEntity entity = response.getEntity();
            if (entity != null) {
                InputStream in = entity.getContent();
                String result = convertStreamToString(in);
                JSONObject json = new JSONObject(result);
            }
        }
    }
}
```

```

        Log.i("JSONClientActivity", "JSON : " +
json.toString());
        JSONArray names = json.names();
        JSONArray values = json.toJSONArray(names);
        for (int i = 0; i < values.length(); i++) {
            Log.i("JSONClientActivity", "Name : " +
names.getString(i)
                    + "\n" + "Value : " +
values.getString(i));
        }
        in.close();
    }
} catch (Exception e) {
    e.printStackTrace();
}
}

```

L'envoi de la commande au moteur de recherche est très simple : il s'agit en effet uniquement d'une banale requête HTTP, méthode GET, les paramètres étant passés dans l'URL. Ils sont au nombre de trois :

- **query** - Ce paramètre est la chaîne de caractères à rechercher.
- **results** - C'est le nombre d'éléments à retourner.
- **output** - C'est le format souhaité pour la réponse. Yahoo! supporte d'ailleurs à la fois JSON et XML.

L'analyse de la réponse reçue du serveur de Yahoo! requiert à peine plus de lignes de code que la requête. Tout d'abord, il faut convertir les octets en chaîne de caractères :

```

private String convertStreamToString(InputStream in) throws IOException {
    try {
        BufferedReader reader = new BufferedReader(new
InputStreamReader(in));
        StringBuilder sb = new StringBuilder();
        String line = null;
        while ((line = reader.readLine()) != null) {
            sb.append(line + "\n");
        }
        return sb.toString();
    } finally {
        try {
            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
}

```

Ensuite, ce String est passé dans le constructeur de la classe *JSONObject* pour y être parsé et obtenir une instance d'objet JSON. Un objet JSON n'est ni plus ni moins qu'un ensemble de paires nom/valeur, les valeurs pouvant être des types simples, d'autres objets ou des tableaux (représentés entre crochets).

Dans le cas présent, l'objet JSON retourné est le suivant :

```
{
  "ResultSet": {
    "Result": [
      {
        "Cache": {
          "Size": "313521",
          "Url": "http://uk.wrs.yahoo.com/_ylt=A0WTeNnQSchKdjKAPE7dmMwF;
_yllu=X3oDMTBwZTdwbWtkBGNvbG8DZQRwb3MDMQRzZWMDc3IEdnRpZAM-/SIG=16iojttgh/
EXP=1244240464/**http%3A//74.6.239.67/search/cache%3Fei=UTF-8%26appid=
YahooDemo%26query=federer%26results=3%26output=json%26u=en.wikipedia.org/
wiki/Roger_Federer%26w=federer%26d=DY0EoxlMS6H0%26icp=1%26.intl=us"
        },
        "ClickUrl": "http://en.wikipedia.org/wiki/Roger_Federer",
        "DisplayUrl": "en.wikipedia.org/wiki/Roger_Federer",
        "MimeType": "text/html",
        "ModificationDate": 1244012400,
        "Summary": "Biography for the Swiss tennis player, Roger Federer.",
        "Title": "Roger Federer - Wikipedia",
        "Url": "http://en.wikipedia.org/wiki/Roger_Federer"
      },
      {
        "Cache": {
          "Size": "18642",
          "Url": "http://uk.wrs.yahoo.com/_ylt=A0WTeNnQSchKdjKAPO7dmMwF;
_yllu=X3oDMTBwbGppbHRwBGNvbG8DZQRwb3MDMGRzZWMDc3IEdnRpZAM-/SIG=164ef116m/
EXP=1244240464/**http%3A//74.6.239.67/search/cache%3Fei=UTF-8%26appid=
YahooDemo%26query=federer%26results=3%26output=json%26u=
www.rogerfederer.com/%26w=federer%26d=YX_Klh1MS58V%26icp=1%26.intl=us"
        },
        "ClickUrl": "http://www.rogerfederer.com/",
        "DisplayUrl": "www.rogerfederer.com/",
        "MimeType": "text/html",
        "ModificationDate": 1244012400,
        "Summary": "Official site of the Swiss tennis player, Roger Federer.",
        "Title": "Roger Federer",
        "Url": "http://www.rogerfederer.com/"
      },
      {
        "Cache": {
          "Size": "167811",
          "Url": "http://uk.wrs.yahoo.com/_ylt=A0WTeNnQSchKdjKAQk7dmMwF;
_yllu=X3oDMTBwbTJyZTk4BGNvbG8DZQRwb3MDMwRzZWMDc3IEdnRpZAM-/SIG=16198e997/
EXP=1244240464/**http%3A//74.6.239.67/search/cache%3Fei=UTF-8%26appid=
YahooDemo%26query=federer%26results=3%26output=json%26u=en.wikipedia.org/
Federer-Nadal_rivalry%26w=federer%26d=UEjABx1MS1Vh%26icp=1%26.intl=us"
        },
        "ClickUrl": "http://en.wikipedia.org/Federer-Nadal_rivalry",
        "DisplayUrl": "en.wikipedia.org/Federer-Nadal_rivalry",
        "MimeType": "text/html",
        "ModificationDate": 1242889200,
        "Summary": "Federer serves to Nadal during the 2008 Wimbledon final.
Roger Federer and Rafael Nadal are professional tennis players engaged in a
...",
        "Title": "Federer128;&#147;Nadal rivalry - Wikipedia, the free
encyclopedia",

```

```

        "Url": "http://en.wikipedia.org/Federer-Nadal_rivalry"
    }
],
    "firstResultPosition": 1,
    "moreSearch":
"/WebSearchService/V1/webSearch?query=federer&appid=YahooDemo&
region=us",
    "totalResultsAvailable": 90200000,
    "totalResultsReturned": 3,
    "type": "web"
}
}

```

Puisque Yahoo! accepte les deux formats, il intéressant de relancer la même requête en changeant juste le paramètre d'output :

<http://search.yahooapis.com/WebSearchService/V1/webSearch?appid=YahooDemo&query=federer&results=3&output=xml>

On obtient alors ceci :

```

<?xml version="1.0" encoding="UTF-8"?>
<ResultSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="urn:yahoo:srch" xsi:schemaLocation="urn:yahoo:srch
http://api.search.yahoo.com/WebSearchService/V1/WebSearchResponse.xsd"
type="web" totalResultsAvailable="98400000" totalResultsReturned="3"
firstResultPosition="1"
moreSearch="/WebSearchService/V1/webSearch?query=federer&appid=
YahooDemo&region=us">
  <Result>
    <Title>Roger Federer - Wikipedia</Title>
    <Summary>Biography for the Swiss tennis player, Roger
Federer.</Summary>
    <Url>http://en.wikipedia.org/wiki/Roger_Federer</Url>
    <ClickUrl>http://en.wikipedia.org/wiki/Roger_Federer</ClickUrl>
    <DisplayUrl>en.wikipedia.org/wiki/Roger_Federer</DisplayUrl>
    <ModificationDate>1244185200</ModificationDate>
    <MimeType>text/html</MimeType>
    <Cache>
      <Url>http://uk.wrs.yahoo.com/_ylt=A0S0mmQ_AcTkgBkBEizdmMwF;
_ylu=X3oDMTBwZTdwbWtkBGNvbG8DZQRwb3MDMQRzZWMDc3IEdnRpZAM-/SIG=16hlnrika/
EXP=1244418495/**http%3A//74.6.239.67/search/cache%3Fei=UTF-8%26appid=
YahooDemo%26query=federer%26results=3%26output=xml%26u=en.wikipedia.org/
wiki/Roger_Federer%26w=federer%26d=DYOExlMS6y8%26icp=1%26.intl=us</Url>
      <Size>324211</Size>
    </Cache>
  </Result>
...

```

On peut effectivement vérifier que l'XML nécessite davantage de caractères pour matérialiser la même information. On peut aussi noter que la réponse XML a été formalisée par le schéma <http://api.search.yahoo.com/WebSearchService/V1/WebSearchResponse.xsd>

Disposer du schéma XML peut s'avérer fort utile, le contrat entre le service et les consommateurs devient alors beaucoup plus formel. En outre, cela permet de transformer le flux XML en une structure objet plus facilement ; de nombreux frameworks de marshallng XML existent et ils génèrent les classes Java équivalentes aux données XML à partir du schéma. Encore faut-il en trouver un qui soit compatible avec Android. En tout cas, JSON ne propose pour l'instant¹ pas de moyen de spécifier un format JSON donné.

Quid de REST ?

REST est l'acronyme de *Representational State Transfer*.

REST n'est pas à mettre sur le même plan que POX ou que JSON.

REST est une architecture logicielle ou plutôt un pattern de communication alors que POX et JSON sont des formats utilisés pour véhiculer de l'information. Dans les applications de type REST, les fonctionnalités et l'état du système sont portés par le concept de ressource.

Chaque ressource REST est identifiée de façon globale et unique par une URI. REST s'attache également à s'appuyer sur les caractéristiques existantes d'HTTP plutôt qu'à redéfinir de nouvelles notions par dessus. Ainsi les méthodes HTTP GET, POST, PUT et DELETE sont respectivement employées pour lire, créer, mettre à jour et supprimer les ressources.

9.2.7 XMPP

XMPP (*eXtensible Messaging and Presence Protocol*) est un protocole ouvert à base d'XML de messagerie instantanée et de présence.

En gagnant en popularité, XMPP a vu ses prérogatives s'élargir pour se transformer en un véritable système Message-oriented middleware (MOM).

C'est ce dernier aspect qui fait de XMPP un protocole très important parfaitement complémentaire aux web services. Effectivement, alors que les web services se consacrent aux communications client/serveur, XMPP, bien que fonctionnant sur une architecture décentralisée de serveurs, est dévolu à la communication entre les clients. C'est donc le maillon qui vient parfaire la chaîne d'outils à disposition du développeur. XMPP offre un cadre à la communication de mobile à mobile, facilitant le développement d'applications de chat bien sûr mais aussi de jeux en réseau, de travail collaboratif en temps réel...

1. Une proposition de spécification pour la définition des données JSON existe bien : <http://json-schema.org/>. Reste à voir si ce projet arrivera à s'imposer.

API XMPP Android

Comme expliqué en introduction, un middleware orienté message convient tout à fait à certains types d'application. Là où les choses se gâtent, c'est qu'après plusieurs rebondissements intervenus au gré des versions successives du SDK, aujourd'hui la dernière version en date du kit de développement, c'est-à-dire la 1.5 r2, n'intègre pas ou plus d'API XMPP !

Les premières versions, comme la m3-rc22 proposaient bien cette API. Ensuite, elle a été renommée en GTalk API sous la justification que son implémentation s'éloignait de plus en plus de la spécification XMPP (par exemple, le protocole sous-jacent au service reposait sur un format binaire et non plus XML) afin de limiter l'usage CPU et donc d'économiser la batterie.

Après, les versions finales du SDK (de la v1.0 jusqu'à la v1.5) avaient purement et simplement supprimé le service GTalk, les raisons invoquées étant que l'API ne présentait pas un niveau de sécurité suffisant. Certains pensent qu'en réalité l'absence de l'API est plus due à une volonté de ne pas concurrencer les systèmes de messageries SMS qui sont très lucratifs pour les opérateurs.

Aujourd'hui, la seule alternative est donc de se tourner vers l'open source, notamment la librairie Smack (<http://www.igniterealtime.org/projects/smack/index.jsp>), qui est un projet porté par la société Jive Software qui édite des logiciels bâtis autour de XMPP.

Malheureusement, comme souvent lorsqu'il s'agit d'importer une librairie non développée spécifiquement pour la plateforme Android, des problèmes de compatibilité apparaissent.

À l'instar de la librairie KSOAP2, il est donc indispensable d'opérer des adaptations particulières pour Android afin de rendre fonctionnelle cette librairie. Ces modifications sont en outre difficiles à gérer car ce fork¹ du projet smack n'est pas officiel et maintenu. De plus, il n'y a aucune garantie que la librairie ainsi modifiée continue de marcher convenablement avec les futures versions du SDK Android.

Une fois importés les packages « `org.jivesoftware.smack.*` », avant toute chose, il faut se connecter au serveur XMPP :

```
private void initConnection() throws XMPPException {
    //Initialisation de la connexion
    ConnectionConfiguration config =
        new ConnectionConfiguration(SERVER_HOST, SERVER_PORT,
SERVICE_NAME);
    m_connection = new XMPPConnection(config);
    m_connection.connect();
    m_connection.login(LOGIN, PASSWORD);
    Presence presence = new Presence(Presence.Type.available);
    m_connection.sendPacket(presence);
    //enregistrement de l'écouteur de messages
    PacketFilter filter = new MessageTypeFilter(Message.Type.chat);
```

1. On désigne par « fork » un projet dérivé d'un autre projet dont il récupère les éléments.

```

        m_connection.addPacketListener(new PacketListener() {
            public void processPacket(Packet packet) {
                Message message = (Message) packet;
                if (message.getBody() != null) {
                    String fromName =
StringUtils.parseBareAddress(message
                                .getFrom());
                    m_discussionThread.add(fromName + ":");
                    m_discussionThread.add(message.getBody());
                    m_handler.post(new Runnable() {
                        public void run() {
                            m_discussionThreadAdapter.notifyDataSetChanged();
                        }
                    });
                }
            }
        }, filter);
    }
}

```

Les constantes définissant les paramètres de connexion ont été spécifiées comme ceci :

```

private final static String SERVER_HOST = "talk.google.com";
private final static int SERVER_PORT = 5222;
private final static String SERVICE_NAME = "gmail.com";
private final static String LOGIN = "florent.garin@gmail.com";
private final static String PASSWORD = "*****";

```

Le serveur XMPP utilisé ici est celui de Google Talk. Le login et le mot de passe correspondent au compte Google avec lequel les messages seront envoyés et reçus.

L'architecture de XMPP est décentralisée, il aurait donc été parfaitement possible d'utiliser un autre serveur XMPP, les serveurs XMPP étant interopérables. La société Jive Software propose le serveur open source Openfire (<http://www.igniterealtime.org/projects/openfire/index.jsp>).

Le reste de la méthode initialisation enregistre un listener afin de réceptionner les messages entrants. Ceux-ci seront affichés à l'écran avec le nom de l'expéditeur.

Cette méthode d'initialisation sera appelée depuis la méthode de lancement de l'activité :

```

private List<String> m_discussionThread;
private ArrayAdapter<String> m_discussionThreadAdapter;
private XMPPConnection m_connection;
private Handler m_handler;
/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    m_handler = new Handler();
    try {
        initConnection();
    } catch (XMPPException e) {

```

```

        e.printStackTrace();
    }
    final EditText recipient = (EditText)
this.findViewById(R.id.recipient);
    final EditText message = (EditText) this.findViewById(R.id.message);
    ListView list = (ListView) this.findViewById(R.id.thread);
    m_discussionThread = new ArrayList<String>();
    m_discussionThreadAdapter = new ArrayAdapter<String>(this,
        R.layout.multi_line_list_item, m_discussionThread);
    list.setAdapter(m_discussionThreadAdapter);
    Button send = (Button) this.findViewById(R.id.send);
    send.setOnClickListener(new View.OnClickListener() {
        public void onClick(View view) {
            String to = recipient.getText().toString();
            String text = message.getText().toString();
            Message msg = new Message(to, Message.Type.chat);
            msg.setBody(text);
            m_connection.sendPacket(msg);
            m_discussionThread.add("moi :");
            m_discussionThread.add(text);
            m_discussionThreadAdapter.notifyDataSetChanged();
        }
    });
}

```

Le message textuel est envoyé à l'adresse XMPP saisie par l'utilisateur.

L'envoi du message se fait suite au clic du bouton « Envoyer ». La ListView retraçant les messages du fil de discussion est alors actualisée.

L'exécution de l'application donne le résultat suivant :



Figure 9.1 — Les messages XMPP

Le correspondant pourra dialoguer en utilisant la même application sur un autre téléphone Android ou depuis un ordinateur à l'aide d'un des nombreux clients XMPP disponibles.

En conclusion, il est relativement simple d'implémenter un client XMPP sur Android. Mais il est très regrettable de ne pas disposer d'une librairie stable et parfaitement adaptée à ce système pour le faire. L'idéal aurait été que cette librairie fasse partie de l'API standard d'Android. Il va falloir se montrer patient, en espérant que Google réintroduise l'API dans les versions futures. En attendant, la librairie Smack, après avoir été un peu modifiée, répond amplement au besoin.

Ce qu'il faut retenir

Pour faire communiquer une application quelconque avec le monde extérieur, le choix en matière de protocole est très vaste. Néanmoins sur Android, compte tenu des contraintes liées à la machine virtuelle Dalvik, ce choix est beaucoup plus limité. Android étant pensé comme un système web, le protocole de transport roi est l'HTTP. Lorsqu'il s'agit d'interroger un serveur distant, un protocole de type REST basé sur de l'XML ou de JSON paraît bien adapté. L'usage de SOAP est bien possible mais la lourdeur de ce protocole ne semble pas le prédisposer aux terminaux mobiles.

Enfin, pour les communications « device to device », le protocole XMPP est une solution sérieuse à envisager. En attendant peut-être que Google fournisse une solution pour l'implémentation de ce type de communication.

10

Sécurité et déploiement

Objectifs

La phase ultime du développement d'une application Android est sa mise à disposition des utilisateurs. Si le codage reste l'essentiel du travail de la réalisation d'une application Android, cette dernière étape ne doit pas être négligée tant les conséquences d'éventuelles maladroites pourraient être difficiles à rattraper.

Ce dernier chapitre traite donc des derniers efforts à fournir avant de voir son application partir vers son public !

10.1 SIGNER LES APPLICATIONS

10.1.1 La phase de développement

Pour pouvoir être déployées, sur un mobile ou sur l'émulateur, toutes les applications Android doivent être signées. Cette règle ne souffre aucune exception.

Cependant, pour faciliter le quotidien du développeur, le *plug-in* Eclipse ADT et les scripts ant générés avec le SDK Android proposent un mode dit debug pour signer les applications. Dans ce mode, le développeur n'aura pas à se soucier de ce problème, tous les aspects liés à la signature de l'application lui sont totalement masqués :

- la génération du certificat autosigné,
- la génération de la paire de clés publique/privée,
- la signature de l'application.

Le certificat et les clés ne seront pas stockés dans les keystores par défaut du système ou de l'utilisateur mais dans un keystore à part créé uniquement pour l'occasion ; il s'agit du fichier « .android/debug.keystore » accessible depuis le répertoire home de l'utilisateur (/home ou C:\Users).

Si le mode debug est très pratique pendant la phase de développement où l'on est amené à modifier, déployer et exécuter le programme de nombreuses fois, il ne convient pas lors de la phase suivante où l'on package l'application en vue des véritables futurs déploiements. En fait, il est strictement impossible de déployer une application signée avec le certificat de debug.

Le mode debug « under the wood »

Quand on regarde de plus près le mode debug, on s'aperçoit que son fonctionnement repose sur les mêmes mécanismes que le mode release, avec des paramètres fixés préalablement.

Le keystore utilisé est « debug.keystore ». Son mot de passe est « android ».

La clé de debug avec laquelle l'application est signée a pour alias « androiddebugkey ».

Le mot de passe est également « android ».

Et le CN (Common Name) du certificat vaut « CN=Android Debug,O=Android,C=US ».

Excepté le keystore, aucun de ces paramètres ne peut être changé, c'est comme cela que le SDK peut exécuter les opérations de signature automatiquement.

10.1.2 La phase de packaging

Lorsque le développement de l'application est achevé, du moins dans sa première version, il faut passer en mode release. Effectivement, aucun terminal n'acceptera d'installer une application signée avec une clé de debug.

Heureusement, un certificat autosigné fera parfaitement l'affaire, ce qui simplifie grandement la procédure et économise les frais d'obtention d'un « vrai » certificat.

Certificat autosigné vs Certificat valide

Les certificats fonctionnent selon un principe d'approbation en cascade. Ainsi, une paire de clés privée/publique ne pourra être réputée comme étant bien la propriété de l'entité représentée par le certificat que si elle a été elle-même signée avec une autre clé privée appartenant cette fois-ci à une organisation faisant foi. Ces organisations sont dénommées Certificat Authority (CA), parmi elles on peut citer Thawte, VeriSign ou Entrust. La validité de leur propre signature est contrôlée grâce à la présence de leur clé publique sur le système. Par exemple, les navigateurs web intègrent une collection importante de CA. Le rôle des CA est de vérifier que l'identité déclarée dans le certificat correspond bien à la réalité. Ce n'est qu'à cette condition qu'ils acceptent d'apposer leur signature engageant ainsi leur crédibilité. C'est de cette activité que les CA tirent leurs revenus.

La différence entre un certificat valide et un certificat autosigné est là : dans le premier cas les informations du certificat ont été vérifiées, et dans le second cas non.

Génération d'un certificat

La première condition à remplir pour signer une application et envisager sa publication est de disposer d'un certificat. Pour cela, il faut utiliser la commande du JDK keytool. Cette commande sert à gérer des sortes de registres de certificats et de clés que l'on appelle *keystores*.

En saisissant la commande suivante, on créera un certificat et une paire de clés :

```
■ keytool -genkeypair -alias for_android -validity 10000
```

« -genkeypair » indique à keytool de générer une paire de clés privé/publique. keytool peut aussi générer des clés symétriques avec l'option -genseckey.

L'option « -alias » précise le nom par lequel le certificat sera référencé dans le keystore.

« -validity » est la validité en jours du certificat. Pour être diffusées sur Android Market, équivalent de l'App Store d'Apple, les applications devront avoir été signées avec un certificat expirant après le 22 octobre 2033. En outre, comme on le verra par la suite, pour faciliter la gestion des mises à jour des applications, il est préférable d'utiliser un certificat avec une validité de longue durée. 10 000 jours, qui équivalent à environ 27 ans, semblent être une bonne valeur.

Suite à l'exécution de la commande, il faudra répondre à une série de questions concernant l'identité à rattacher au certificat.

```
Tapez le mot de passe du Keystore :
Quels sont vos prénom et nom ?
[Unknown] : Florent Garin
Quel est le nom de votre unité organisationnelle ?
[Unknown] : florentgarin.org
Quel est le nom de votre organisation ?
[Unknown] : florentgarin.org
Quel est le nom de votre ville de résidence ?
[Unknown] : Toulouse
Quel est le nom de votre état ou province ?
[Unknown] :
Quel est le code de pays à deux lettres pour cette unité ?
[Unknown] : FR
Est-ce CN=Florent Garin, OU=florentgarin.org, O=florentgarin.org, L=Toulouse,
ST
=Unknown, C=FR ?
[non] : oui
Spécifiez le mot de passe de la clé pour <for_android>
(appuyez sur Entrée s'il s'agit du mot de passe du Keystore) :
```

Il est toutefois possible d'éviter cette interaction en mentionnant dans la ligne de commande avec l'option « -dname "CN=Florent Garin, OU=florentgarin.org, O=florentgarin.org, L=Toulouse, ST" » (distinguished name) les réponses fournies.

Ici dans l'exemple, il n'a pas été précisé de fichier keystore (option -keystore) : la valeur par défaut sera donc utilisée. Il s'agit du fichier .keystore stocké dans le répertoire home de l'utilisateur. On peut bien sûr spécifier un autre chemin et créer autant de fichiers keystore que l'on souhaite.

Pour avoir une description exhaustive de l'utilitaire `keytool`, il suffit de se rendre sur la page de la documentation du JDK consacrée à la commande :

- <http://java.sun.com/javase/6/docs/technotes/tools/windows/keytool.html>
(Windows)
- <http://java.sun.com/javase/6/docs/technotes/tools/solaris/keytool.html>
(Unix)

Le certificat en main, on va maintenant pouvoir signer l'application.

Attention, il faut travailler à partir du fichier `apk` de l'application dans sa version non signée. On ne peut pas, par exemple, récupérer le fichier `apk` qui se trouve dans le répertoire `bin`. Celui-ci a été créé par Eclipse lorsqu'on a lancé l'application ; il a donc déjà été signé avec le certificat de debug pour pouvoir être déployé sur l'émulateur.

Pour obtenir l'application packagée mais non signée, il faut faire un clic droit dans Eclipse sur le projet puis « Android Tools > Export Unsigned Application Package ».

Pour ceux qui n'utilisent pas Eclipse, le fichier `ant` de compilation « `build.xml` » possède une tâche spécialement prévue pour cela. En effet, en tapant la commande :

```
■ ant release
```

la signature effective de l'application peut maintenant se faire grâce à l'outil `jarsigner` qui, à l'instar de `keytool`, est une commande standard du JDK.

`keytool` et `jarsigner` sont deux commandes présentes dans le répertoire `bin` du JDK, celui-là même où se trouvent les exécutables `java`, `jar`, `javac`... Il faudra donc vérifier que ce répertoire est bien dans le `PATH`.

Pour signer l'application, il faut alors saisir la commande suivante :

```
■ jarsigner -verbose SampleAppWidget.apk for_android
```

L'option facultative « `-verbose` » (bavard en français) permet de suivre et de comprendre précisément les transformations que la commande apporte au fichier `apk` :

```
Enter Passphrase for keystore:
  adding: META-INF/MANIFEST.MF
  adding: META-INF/FOR_ANDROID.SF
  adding: META-INF/FOR_ANDROID.DSA
signing: res/drawable/appwidget_bg_portrait.png
signing: res/drawable/icon.png
signing: res/layout/sample_appwidget_layout.xml
signing: res/xml/sample_appwidget_info.xml
signing: AndroidManifest.xml
signing: resources.arsc
signing: classes.dex
```

La signature est faite avec la clé ayant pour alias `for_android`. Comme avec `keytool`, on peut préciser un autre keystore que celui par défaut.

Pour vérifier qu'un fichier apk a été correctement signé ou pour connaître le certificat qui a été utilisé, on peut se servir de l'option `-verify` accompagné de `-certs` :

```
jarsigner -verify -verbose -certs SampleAppWidget.apk
```

Pour une vue complète ; la page de manuel de jarsigner est disponible sur la documentation officielle du JDK :

- <http://java.sun.com/javase/6/docs/technotes/tools/windows/jarsigner.html> (Windows)
- <http://java.sun.com/javase/6/docs/technotes/tools/solaris/jarsigner.html> (Unix)

Toutefois, la vérification la plus probante est encore de déployer l'application avec la commande `adb` sur un émulateur ou encore mieux sur un vrai téléphone :

```
adb install SampleAppWidget.apk
```

Le résultat du déploiement s'affiche en sortie de commande :

```
79 KB/s (13993 bytes in 0.172s)
  pkg: /data/local/tmp/SampleAppWidget.apk
Success
```

Toutefois, il ne faut pas se contenter du « Success », il est plus que préconisé de retester entièrement l'application. On peut avoir des surprises, par exemple la clé d'utilisation de Google Maps est liée au certificat. En passant du certificat de debug au certificat « release », il ne faudra pas oublier de récupérer une nouvelle clé et de mettre à jour le code.

On sait dorénavant comment générer un certificat et signer l'application de sorte qu'elle puisse être installée sur les terminaux Android.

Il faut maintenant organiser sa diffusion et gérer ses évolutions futures.

Signature et permissions

Android attribue certaines permissions en fonction de la signature de l'application. Par exemple deux programmes distincts mais signés avec le même certificat pourront être exécutés dans le même processus. Cette caractéristique permet d'envisager de ventiler les composants d'un logiciel (les services, les activités...) dans plusieurs applications android (modules) pour les gérer de façon autonome (les déploiements, les mises à jour...).

En outre, les composants android peuvent accorder aux autres applications le droit de les solliciter en fonction de leur signature.

10.2 PUBLIER SON APPLICATION

10.2.1 Gestion des versions

Lorsque l'application est prévue pour être installée sur un parc de téléphones important, voire pour une diffusion publique, une gestion de version applicative devient indispensable.

Il faut en effet, informer explicitement les utilisateurs, par l'ajout de métadonnées directement dans le fichier apk, du numéro de version exact de l'application. Ces données sont non seulement utiles aux propriétaires de smartphones, pour qu'ils puissent faire un choix quant à l'éventuelle mise à jour de leur combiné, mais aussi aux autres applications qui pourraient dépendre du programme. Ainsi ces applications peuvent contrôler que les composants sur lesquels elles s'appuient sont bien présents sur le système Android et dans une version compatible.

Pour déclarer la version des applications, Android propose deux attributs, `android:versionCode` et `android:versionName` positionnés sur le tag racine du fichier manifeste :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.florentgarin.android.appwidget" android:versionCode="1"
    android:versionName="2.6.30">
    ...
```

Ces attributs n'ont pas la même finalité :

- **android:versionCode** – Cet attribut est purement technique, il s'agit d'un numéro servant de comparaison, pour déterminer laquelle des deux versions d'une même application est la plus récente. La meilleure stratégie est d'incrémenter ce numéro d'une unité à chaque publication.
- **android:versionName** – Cet attribut est destiné à être communiqué aux utilisateurs. La gestion de cet attribut de type chaîne de caractères est libre et de la responsabilité des développeurs de l'application. L'usage courant qui en est fait est d'utiliser le format « version_majeur.version_mineur.x » ; par exemple 2.6.30, le troisième chiffre indiquant une révision suite à une correction de bug.

Pour réaliser une mise à jour, l'application devra donc déclarer dans son manifeste un attribut `versionCode` supérieur à celui précédemment défini. Ce n'est pas la seule condition à remplir pour faire des « updates » : une autre exigence incontournable est de signer le nouveau fichier apk avec la même signature que celle utilisée pour la version précédente.

Si hélas le fichier keystore contenant la clé privée a été perdu, le fournisseur de l'application sera dans l'incapacité de signer la nouvelle version avec le bon certificat, il n'aura donc pas d'autre choix que de renommer l'application en changeant son identifiant qui est matérialisé par son package Java.

On touche ici du doigt l'importance de bien conserver le fichier keystore. Si le perdre est problématique pour la gestion des versions, se le faire dérober est pire. En effet, quiconque possède un certificat peut se faire passer pour son propriétaire et publier de « fausses » versions de ses applications.

De plus, même si les données d'identification du certificat, le distinguished name (DN), ne sont pas tenues d'être vérifiées par une autorité, une fois l'application diffusée, il ne sera pas possible de les modifier sans régénérer une clé. Il est donc conseillé de bien choisir le DN.

10.2.2 Android Market

L'Android market est le système de publication d'applications officiel d'Android. Néanmoins, contrairement à l'App Store d'Apple, la distribution d'applications au public n'est pas un monopole de Google et techniquement, rien n'empêche d'installer des applications depuis une autre source. D'ailleurs, des alternatives à Android Market, comme par exemple SlideMe (<http://slideme.org>), émergent peu à peu.

Pour proposer son application sur Android Market (<http://market.android.com/publish>), il faudra détenir un compte Google et s'acquitter des frais d'enregistrement qui s'élèvent à 15 \$.

Bien sûr, l'application devra être parfaitement packagée et signée, par ailleurs Android Market requiert que la date de validité du certificat soit postérieure au 22 octobre 2033, qu'une icône et un intitulé soient précisés dans le manifeste (attributs `android:icon` et `android:label`) et que les champs de version (`android:versionCode` et `android:versionName`) soient renseignés.

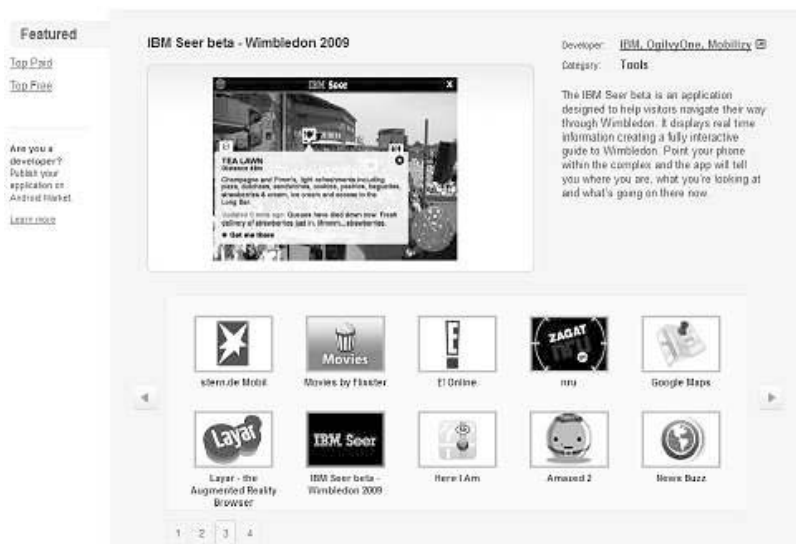


Figure 10.1 — Le site web de l'Android Market

Le système Android inclut l'application client de l'Android Market (uniquement sur les téléphones et non sur l'émulateur) grâce à laquelle on peut parcourir le dépôt d'applications, explorer les différentes catégories ou même effectuer des recherches.

L'application Market peut se lancer par du code en invoquant la méthode `startActivity` avec un Intent de type `ACTION_VIEW` :

```
intent intent = new Intent();
intent.setAction(Intent.ACTION_VIEW);
intent.setData(Uri.parse("http://market.android.com/search?q=pub:Google"));
startActivity(intent);
```

On peut préciser dans l'intent une url contenant des éléments de recherche. Cette fonctionnalité est très utile pour, par exemple, inclure dans l'application un lien vers d'autres logiciels du même éditeur, des modules supplémentaires ou des mises à jour.

Ce qu'il faut retenir

Même si l'impatience du développeur peut être grande au moment de la publication de son application, il est souhaitable de ne pas brûler les dernières étapes au risque de ruiner sa réputation.

On ne répétera jamais assez l'importance qu'il y a à effectuer une batterie de tests très poussée de l'application finalisée et packagée sur l'ensemble des terminaux pour lesquels elle est destinée.

Annexe

Usage commande aapt

Android Asset Packaging Tool

Usage:

```
aapt l[list] [-v] [-a] file.{zip,jar,apk}
    List contents of Zip-compatible archive.
aapt d[ump] WHAT file.{apk} [asset [asset ...]]
    badging          Print the label and icon for the app declared in APK.
    permissions      Print the permissions from the APK.
    resources        Print the resource table from the APK.
    configurations   Print the configurations in the APK.
    xmltree          Print the compiled xmls in the given assets.
    xmlstrings       Print the strings of the given compiled xml assets.
aapt p[ackage] [-d][-f][-m][-u][-v][-x][-z][-M AndroidManifest.xml] \
    [-O extension [-O extension ...]] \
    [-g tolerance] \
    [-j jarfile] \
    [-I base-package [-I base-package ...]] \
    [-A asset-source-dir] [-P public-definitions-file] \
    [-S resource-sources [-S resource-sources ...]]
    [-F apk-file] [-J R-file-dir] \
    [raw-files-dir [raw-files-dir] ...]
    Package the android resources. It will read assets and resources
    that are supplied with the -M -A -S or raw-files-dir arguments.
    The -J -P -F and --R options control which files are output.
aapt r[emove] [-v] file.{zip,jar,apk} file1 [file2 ...]
    Delete specified files from Zip-compatible archive.
aapt a[dd] [-v] file.{zip,jar,apk} file1 [file2 ...]
    Add specified files to Zip-compatible archive.
aapt v[ersion]
    Print program version.
Modifiers:
-a print Android-specific data (resources, manifest) when listing
-c specify which configurations to include. The default is all
  configurations. The value of the parameter should be a comma
  separated list of configuration values. Locales should be specified
  as either a language or language-region pair. Some examples:
    en
    port,en
    port,land,en_US
```

If you put the special locale, `zz_ZZ` on the list, it will perform pseudolocalization on the default locale, modifying all of the strings so you can look for strings that missed the internationalization process. For example:

```
port,land,zz_ZZ
-d one or more device assets to include, separated by commas
-f force overwrite of existing files
-g specify a pixel tolerance to force images to grayscale, default 0
-j specify a jar or zip file containing classes to include
-m make package directories under location specified by -J
-u update existing packages (add new, replace older,
                           remove deleted files)
-v verbose output
-x create extending (non-application) resource IDs
-z require localization of resource attributes marked with
  localization="suggested"
-A additional directory in which to find raw asset files
-F specify the apk file to output
-I add an existing package to base include set
-J specify where to output R.java resource constant definitions
-M specify full path to AndroidManifest.xml to include in zip
-P specify where to output public resource definitions
-S directory in which to find resources. Multiple directories
  will be scanned and the first match found (left to right)
  will take precedence. -0 specifies an additional extension
  for which such files will not be stored compressed in the .apk.
  An empty string means to not compress any files at all.
```

Usage commande adb

Android Debug Bridge version 1.0.20

```
-d - directs command to the only connected USB device
    returns an error if more than one USB device is present.
-e - directs command to the only running emulator.
    returns an error if more than one emulator is running.
-s <serial number> - directs command to the USB device or emulator with
                   the given serial number
-p <product name or path> - simple product name like 'sooner', or
                           a relative/absolute path to a product out
                           directory like 'out/target/product/sooner'.
                           If -p is not specified, the
                           ANDROID_PRODUCT_OUT environment variable is
                           used, which must be an absolute path.
devices - list all connected devices
device commands:
adb push <local> <remote> - copy file/dir to device
adb pull <remote> <local> - copy file/dir from device
adb sync [ <directory> ] - copy host->device only if changed
                           (see 'adb help all')
adb shell - run remote shell interactively
adb shell <command> - run remote shell command
adb emu <command> - run emulator console command
adb logcat [ <filter-spec> ] - View device log
adb forward <local> <remote> - forward socket connections
                               forward specs are one of:
                               tcp:<port>
```

```

                                localabstract:<unix domain socket name>
                                localreserved:<unix domain socket name>
                                localfilesystem:<unix domain socket name>
                                dev:<character device name>
                                jdwp:<process pid> (remote only)
adb jdwp                        - list PIDs of processes hosting
                                a JDWP transport
adb install [-l] [-r] <file> - push this package file to the device
                                and install it
                                ('-l' means forward-lock the app)
                                ('-r' means reinstall the app,
                                keeping its data)
adb uninstall [-k] <package> - remove this app package from the device
                                ('-k' means keep the data
                                and cache directories)
adb bugreport                  - return all information from the device
                                that should be included in a bug report.
adb help                       - show this help message
adb version                    - show version num
DATAOPTS:
(no option)                    - don't touch the data partition
-w                              - wipe the data partition
-d                              - flash the data partition
scripting:
adb wait-for-device            - block until device is online
adb start-server               - ensure that there is a server running
adb kill-server                - kill the server if it is running
adb get-state                  - prints: offline | bootloader | device
adb get-serialno               - prints: <serial-number>
adb status-window              - continuously print device status
                                for a specified device
adb remount                    - remounts the /system partition
                                on the device read-write
adb root                       - restarts adb with root permissions
networking:
adb ppp <tty> [parameters]    - Run PPP over USB.
Note: you should not automatically start a PDP connection.
<tty> refers to the tty for PPP stream. Eg. dev:/dev/omap_csmtt1
[parameters] - Eg. defaultroute debug dump local notty usepeerdns
adb sync notes: adb sync [ <directory> ]
<localdir> can be interpreted in several ways:
- If <directory> is not specified, both /system and /data partitions
  will be updated.
- If it is "system" or "data", only the corresponding partition
  is updated.

```

Usage commande android

```

Usage:
  android [global options] action [action options]
Global options:
-h --help      This help.
-s --silent    Silent mode: only errors are printed out.
-v --verbose   Verbose mode: errors, warnings
                                and informational messages are printed.
Valid actions are composed of a verb and an optional direct object:

```

```

- list      : Lists existing targets or virtual devices.
- list avd  : Lists existing Android Virtual Devices.
- list target : Lists existing targets.
- create avd : Creates a new Android Virtual Device.
- move avd   : Moves or renames an Android Virtual Device.
- delete avd : Deletes an Android Virtual Device.
- update avd : Updates an Android Virtual Device to match
               the folders of a new SDK.
- create project: Creates a new Android Project.
- update project: Updates an Android Project (must have
                 an AndroidManifest.xml).

Action "list ":
  Lists existing targets or virtual devices.
Options:
  No options
Action "list avd":
  Lists existing Android Virtual Devices.
Options:
  No options
Action "list target":
  Lists existing targets.
Options:
  No options
Action "create avd":
  Creates a new Android Virtual Device.
Options:
  -t --target  Target id of the new AVD [required]
  -c --sdcard  Path to a shared SD card image, or size of
               a new sdcard for the new AVD
  -p --path    Location path of the directory where
               the new AVD will be created
  -n --name    Name of the new AVD [required]
  -f --force   Force creation (override an existing AVD)
  -s --skin    Skin of the new AVD
Action "move avd":
  Moves or renames an Android Virtual Device.
Options:
  -p --path    New location path of the directory where to move the AVD
  -n --name    Name of the AVD to move or rename [required]
  -r --rename  New name of the AVD to rename
Action "delete avd":
  Deletes an Android Virtual Device.
Options:
  -n --name    Name of the AVD to delete [required]
Action "update avd":
  Updates an Android Virtual Device to match the folders of a new SDK.
Options:
  -n --name    Name of the AVD to update [required]
Action "create project":
  Creates a new Android Project.
Options:
  -k --package Package name [required]
  -n --name    Project name
  -a --activity Activity name [required]
  -t --target  Target id of the new project [required]
  -p --path    Location path of new project [required]
Action "update project":

```

```

    Updates an Android Project (must have an AndroidManifest.xml).
Options:
  -t --target    Target id to set for the project [required]
  -p --path      Location path of the project [required]
  -n --name      Project name

```

Usage commande aidl

```

usage: aidl OPTIONS INPUT [OUTPUT]
       aidl --preprocess OUTPUT INPUT...
OPTIONS:
  -I<DIR>      search path for import statements.
  -d<FILE>      generate dependency file.
  -p<FILE>      file created by --preprocess to import.
  -o<FOLDER>    base output folder for generated files.
  -b           fail when trying to compile a parcelable.
INPUT:
  An aidl interface file.
OUTPUT:
  The generated interface files.
  If omitted and the -o option is not used, the input filename
  is used, with the .aidl extension changed to a .java extension.
  If the -o option is used, the generated files will be placed
  in the base output folder, under their package folder

```

Usage commande dx

```

usage:
dx --dex [--debug] [--verbose] [--positions=<style>] [--no-locals]
  [--no-optimize] [--statistics] [--[no-]optimize-list=<file>] [--no-strict]
  [--keep-classes] [--output=<file>] [--dump-to=<file>] [--dump-width=<n>]
  [--dump-method=<name>[*]] [--verbose-dump] [--no-files] [--core-library]
  [<file>.class | <file>.{zip,jar,apk} | <directory>] ...
  Convert a set of classfiles into a dex file, optionally embedded in a
  jar/zip. Output name must end with one of: .dex .jar .zip .apk.
  Positions options: none, important, lines.
dx --annotool --annotation=<class> [--element=<element types>]
  [--print=<print types>]
dx --dump [--debug] [--strict] [--bytes] [--basic-blocks | --rop-blocks]
  [--width=<n>] [<file>.class | <file>.txt] ...
  Dump classfiles in a human-oriented format.
dx --junit [-wait] <TestClass>
  Run the indicated unit test.
dx -J<option> ... <arguments, in one of the above forms>
  Pass VM-specific options to the virtual machine that runs dx.
dx --version
  Print the version of this tool (1.2).
dx --help
  Print this message.

```

Usage commande emulator

use `'-avd <name>'` to start the emulator program with a given Android Virtual Device (a.k.a. AVD), where `<name>` must correspond to the name of one of the existing AVDs available on your host machine.
See `-help-virtual-device` to learn how to create/list/manage AVDs.
As a special convenience, using `'@<name>'` is equivalent to using `'-avd <name>'`.

Usage commande mksdcard

mksdcard: create a blank FAT32 image to be used with the Android emulator
usage: mksdcard [-l label] <size> <file>
if <size> is a simple integer, it specifies a size in bytes
if <size> is an integer followed by 'K', it specifies a size in KiB
if <size> is an integer followed by 'M', it specifies a size in MiB

Usage commande apkbuilder

A command line tool to package an Android application from various sources.
Usage: apkbuilder <out archive> [-v][-u][-storetype STORE_TYPE]
[-z inputzip] [-f inputfile] [-rf input-folder] [-rj -input-path]
-v Verbose.
-u Creates an unsigned package.
-storetype Forces the KeyStore type. If omitted the default is used.
-z Followed by the path to a zip archive.
Adds the content of the application package.
-f Followed by the path to a file.
Adds the file to the application package.
-rf Followed by the path to a source folder.
Adds the java resources found in that folder to the application package, while keeping their path relative to the source folder.
-rj Followed by the path to a jar file or a folder containing jar files.
Adds the java resources found in the jar file(s) to the application package.
-nf Followed by the root folder containing native libraries to include in the application package.

Usage commande sqlite3

```
SQLite version 3.5.9
Enter ".help" for instructions
sqlite> .help
.bail ON|OFF      Stop after hitting an error.  Default OFF
.databases        List names and files of attached databases
.dump ?TABLE? ... Dump the database in an SQL text format
.echo ON|OFF      Turn command echo on or off
.exit            Exit this program
.explain ON|OFF   Turn output mode suitable for EXPLAIN on or off.
.header(s) ON|OFF Turn display of headers on or off
.help            Show this message
```

<code>.import FILE TABLE</code>	Import data from FILE into TABLE
<code>.indices TABLE</code>	Show names of all indices on TABLE
<code>.load FILE ?ENTRY?</code>	Load an extension library
<code>.mode MODE ?TABLE?</code>	Set output mode where MODE is one of: <code>csv</code> Comma-separated values <code>column</code> Left-aligned columns. (See <code>.width</code>) <code>html</code> HTML <code><table></code> code <code>insert</code> SQL insert statements for TABLE <code>line</code> One value per line <code>list</code> Values delimited by <code>.separator</code> string <code>tabs</code> Tab-separated values <code>tcl</code> TCL list elements
<code>.nullvalue STRING</code>	Print STRING in place of NULL values
<code>.output FILENAME</code>	Send output to FILENAME
<code>.output stdout</code>	Send output to the screen
<code>.prompt MAIN CONTINUE</code>	Replace the standard prompts
<code>.quit</code>	Exit this program
<code>.read FILENAME</code>	Execute SQL in FILENAME
<code>.schema ?TABLE?</code>	Show the CREATE statements
<code>.separator STRING</code>	Change separator used by output mode and <code>.import</code>
<code>.show</code>	Show the current values for various settings
<code>.tables ?PATTERN?</code>	List names of tables matching a LIKE pattern
<code>.timeout MS</code>	Try opening locked tables for MS milliseconds
<code>.width NUM NUM ...</code>	Set column widths for "column" mode

Index

Symboles

2D/3D 134
3G 164

A

AAC 14
aapt 24
AccelerateDecelerateInterpolator 122
AccelerateInterpolator 122
accéléromètre 174
Access Linux Plateform (ALP) 11
Activité (Activity) 28, 34, 66, 74, 132
activitycreator 23
Adapter 59
adb 24
Adobe
 Air 10
aidl 24
Air 10
AJAX (Asynchronous JavaScript and XML) 42
 GWT 15
AlertDialog 133
AMR-NB 169
AnalogClock 68
Android Asset Packaging Tool (aapt) 24, 38
Android Debug Bridge (adb) 24

Android Interface Definition Language (aidl) 24, 81
Android Market 197, 201
Android Virtual Devices (AVD) 23, 157, 170
android.jar 22
AndroidManifest.xml 30
anim 37
animation 121
AnimationSet 122
ANR (Application Not Responding) 95
Apache open source licence v2 3
API
 cartographie 156
 DOM (Document Object Model) 42
 HttpClient 182
 JAXB 16
 JDBC 106, 110, 178
 Media 168
 OpenGL ES 1.0 137
 réseau bas niveau 164
 SQLite 106
apk 23, 198
apkbuilder 25
App Widget 138
appareil photo numérique (APN) 166
Apple 5
 App Store 6, 201
 iPhone 6, 42

- iTunes 6
- application (publier) 200
- AppWidgetProvider 139, 140
- Archos 3
- assets 38
- Asus 2
- AttributeSet 121
- audio
 - AMR-NB 169
 - format audio 14
 - MIDI 172
 - WMA 169
- AutoCompleteTextView 59

B

- barre de notification 130
- barre de statut 130
- BlackBerry 8
- Blob 114
- Bluetooth 6, 166
- boîte de dialogue 131
- BroadcastReceiver 34, 74, 90, 139, 140
- Bundle 93, 151
- bus
 - message 90
- Button 56

C

- Camera 166
- Canonical Ltd 11
 - Ubuntu Mobile 11
- carte SD 100
- cartographie 156
- certificat
 - autosigné 195
 - Certificat Authority (CA) 196
 - génération 197
- CheckBox 53
- classe
 - R 38
 - Vibrator 172

- ClassLoader 15, 102
- clé
 - debug key 25
 - keystore 196
 - publique/privée 195
- codec 170
- communication 79
- composant 73
 - étendre 120
- composant graphique
 - AnalogClock 68
 - AutoCompleteTextView 59
 - barre de statut 130
 - boîte de dialogue 131
 - Button 56
 - CheckBox 53
 - Context Menu 120
 - DatePicker 60
 - DigitalClock 68
 - EditText 52
 - Gallery 61
 - ImageButton 65
 - ImageView 65
 - Option Menu 118
 - ProgressBar 66
 - RadioGroup 55
 - RatingBar 69
 - Spinner 58
 - TextView 51
 - TimePicker 61
 - Toast 128
 - ToggleButton 54
- ConnectivityManager 164
- constructeur
 - Apple 5
 - Archos 3
 - Asus 2
 - BlackBerry 8
 - HTC 2
 - Huawei 2
 - Intel 1
 - LG 2

- Motorola 1
- Nokia 5, 7
- nVidia 1
- Palm 9
- Samsung 1
- Sony Ericsson 1
- Sun Microsystems 10
- Toshiba 2
- ContentProvider 34, 74, 99, 108, 112
- ContentResolver 108, 111, 114
- Context 90
- Context Menu 120
- Corba 81, 87
 - IDL 24
 - IIOP 180
- Core Applications 3
- CSS 9, 145
- Cursor 109, 114
- CycleInterpolator 122

D

- Dalvik 15
- Dalvik Debug Monitor Service (DDMS) 27, 94
- DatePicker 60
- DatePickerDialog 60, 133
- debug 196
- debug key 25
- DecelerateInterpolator 122
- dex (Dalvik Executable) 25, 180
- DHCP 165
- Dialog 133
- DigitalClock 68
- DIP (Density-independent Pixels) 140
- Direct3D 14
- DOM (Document Object Model) 38, 42
- donnée
 - partager 108
 - persistance 99
- draw9patch 136
- Drawable 37, 40, 63, 124, 134
- dx 25

E

- eBay 1
- Eclipse 26
 - Plug-in 16, 23, 26, 195
- Eclipse Public License 7
- écran tactile 173
- EDGE 164
- EditText 52
- émulateur Android 23
- environnement d'exécution 15
- Exchange ActiveSync 7

F

- fichier
 - écriture 101
 - lecture 100
 - ressource 124
- Firefox 14
- Flash 10
- Flex 10, 48
- FrameLayout 71, 139

G

- Gallery 61
- Garbage Collector 27
- Garbage Collector (GC) 20
- Garmin 1
- Gecko 14
- géolocalisation 154
- GeoPoint 161
- GIF 134
- GLSurfaceView 138
- Google 1
 - Google Maps 4, 154, 163
- GPS 154
 - Garmin 1
- GTalk 190
- GWT (Google Web Toolkit) 15

H

H.263 169
H.264 14
Handler 67
HFP (Hands-Free Profile) 166
HSP (Headset Profile) 166
HTC 2
HTML 9, 145, 177
Huawei 2

I

IBinder 84
id 40
IDL 24
image
 AVD 157
 Drawable 37
 format d'image 14, 134
ImageAdapter 62
ImageButton 65
ImageView 62, 65, 123, 135
instrumentation 35
Intel 1
Intent 74, 87, 90, 131, 149
IntentFilter 76, 92
interface
 graphique 45
 homme/machine (IHM) 45, 117
interpolator 122
IPC (Inter-Process Communication) 115
iPhone 6, 42
iTunes 6

J

J2ME 137
Java 13
 Swing 16
Java Development Kit (JDK) 16, 26
javadoc 22
JavaFx 10, 48
JavaME (Java Micro Edition) 11

JavaScript 9, 42, 177
JavaSE (Java Standard Edition) 11
JAXB 16
JDBC 106, 110, 178
JET Creator 172
JET Engine 172
JPEG 14, 134
JSON (JavaScript Object Notation) 9, 184
JSONObject 186
JSR 239 137
JUnit 35, 174

K

keystore 196
keytool (JDK) 197
KSOAP2 182

L

langue 125
layout 37, 40, 70, 117
 FrameLayout 71
 LinearLayout 71
 ListView 70
 RelativeLayout 72
 TableLayout 71
LayoutAnimationController 124
LayoutInflater 129
LayoutParams 50
LevelListDrawable 134
LG 2
libc 13
librairie
 android.jar 22
 libc 13
 rt.jar 16
licence
 Apache open source licence v2 3
 Eclipse Public License 7
LinearInterpolator 122
LinearLayout 47, 50, 71, 139

Linux 11, 13
ListAdapter 59, 70
ListView 70
LocationManager 155

M

machine virtuelle
 Dalvik 15
 registered-based 15
 stack-based 15
manifeste 30, 111, 114
MapActivity 161
MapController 161
MapView 160, 161
Market
 Android Market 201
Media 168
MediaPlayer 169
MediaRecorder 170
MediaStore 171
menu 38, 117
 Context Menu 120
 Option Menu 118
MenuItem 119
message
 bus 90
middleware 15, 180
 MOM (Message-oriented
 middleware) 189
MIDI 172
MIME 76, 113
mode
 debug 196
module 73
Mojo SDK 9
MOM (Message-oriented middleware)
 189
MotionEvent 173
Motorola 1
Mozilla Firefox 14
MP3 14
MPEG-4 14

MTOM 183
MySQL 179

N

NetShare 6
NinePatchDrawable 136
Nokia 5, 7, 42
Notification 131
NotificationManager 131
NTT DoCoMo 1
nVidia 1

O

Ogg Vorbis 14
onDraw 120
onMeasure 120
Open Handset Alliance (OHA) 1, 7
OpenGL 14
OpenGL 1.3 137
OpenGL ES 1.0 14, 137
opérateur
 NTT DoCoMo 1
 Sprint 1
 T-Mobile 1
Option Menu 118
Oracle 10
Overlay 162

P

package
 apk 23
Palm 9
 Palm Pré 9, 42
Parcel 84
Parcelable 82, 88
PDU (Protocol Description Unit) 151
PendingIntent 131
permission 33, 111
persistance 99
Plug-in Eclipse 16, 23, 26, 195

- PNG 14, 134
- POX (Plain Old XML) 184
- PreferenceActivity 105
- PreferenceManager 105
- préférences
 - écriture 103
 - lecture 103
 - utilisateur 102
- Process 33
- programmation
 - enum 21
- ProgressBar 66
- ProgressDialog 133
- Projection 163
- puce
 - Intel 1
 - nVidia 1
- push mail 7

R

- R (classe) 38
- RadioButton 55
- RadioGroup 55
- ramasse-miettes 132
- RatingBar 69
- raw 38, 170
- registered-based 15
- RelativeLayout 72, 139
- RemoteViews 131, 139
- répertoire
 - add-ons 22
 - anim 37, 121
 - layout 37
 - menu 38
 - platforms 22
 - raw 38
 - samples 22
 - tools 22
 - usb_driver 22
 - values 37, 64, 145
 - xml 38
- réseau 177

- réseau bas niveau 164
- ResourceBundle 124
- ressources 37, 124
 - assets 38
- REST (REpresentational State Transfer) 108, 189
- RIA (Rich Internet Applications) 10
- RIM (Research In Motion) 8
- RMI 81
 - JRMP 180
- RPC 181
- rt.jar 16

S

- Samsung 1
- SAX 38
- Scripts Ant 26
- SD (carte) 100
- serveur
 - DHCP 165
- service 81
- Service 34, 74
- Shape 136
- ShapeDrawable 136
- SharedPreferences 102
- signature 199
- Silverlight 10, 48
- SMS 27, 92, 151
- SmsManager 152
- SOAP (Simple Object Access Protocol) 88, 181
- Software Development Kit (SDK) 22, 26
- Sony Ericsson 1
- Spinner 58
- Sprint 1
- SQLite 14, 106
- sqlite3 25, 107
- SQLiteDatabase 112
- SQLiteOpenHelper 106
- stack-based 15
- style 145

Sun Microsystems

- JavaFx 10
- SurfaceHolder 167
- SurfaceView 121, 167
- Swing 16, 57, 59, 66
- Symbian OS 3, 7
- synthetic methods 21
- système
 - de fichiers 100
- système d'exploitation
 - Symbian OS 3, 7
 - WebOS 9
 - Windows Mobile 3, 7

T

- T-Mobile 1
- TableLayout 71
- Task 144
- Task affinity 33
- téléphonie 149
- TelephonyManager 164
- TextView 47, 51
- thème 33, 145
- thread 66, 87, 96
- TimePicker 61
- TimePickerDialog 133
- Toast 94, 128
- ToggleButton 54
- Toshiba 2
- TouchUtils 174

U

- Ubuntu Mobile 11
- URI 77, 108
- UriMatcher 113
- utilisateur 128
 - préférences 102

V

- values 37, 64
- VB 7
- version (gestion) 200
- Vibrator 172
- vibreux 172
- vidéo
 - format vidéo 14
 - H.263 169
 - WMV 169
- VideoView 169
- View 45, 62
- ViewGroup 45, 70, 124
- Visual Studio 7

W

- web service 181
 - SOAP 88
- webapp 41
- WebKit 5, 14, 42, 177
- WebOS 9
- WebView 177
- Wi-Fi 165
- widget Voir composant graphique
 - App Widget 138
- WifiManager 165
- WiMo 7
- Windows CE 7
- Windows Mobile 3, 7
- Windows Pocket PC 7
- WMA 169
- WMV 169
- WSDL (Web Services Description Language) 181

X

- XML 9
- XMPP (eXtensible Messaging and Presence Protocol) 189
- XUL 14