Développement d'applications mobiles - Android
Partie 1

Présentation de la plateforme Android

1) Qu'est-ce qu'Android ?

Android est une combinaison de trois composants :

- un **système d'exploitation** open-source libre pour appareils mobiles. Ce système repose sur un noyau Linux.
- une **plateforme de développement** open-source pour la création d'applications pour mobiles
- des **équipements**, en particulier des téléphones portables, utilisant le système d'exploitation Android ainsi que les applications développées pour lui.

Cette plateforme est open source, principalement sous licence Apache 2.0. Par contre, les mises à jour du noyau Linux utilisent la licence GPLv2. Le développement d'applications s'appuie sur le langage Java.

Elle fonctionne sur les mobiles, les tablettes et les systèmes embarqués, et objets connectés en général (montres, lunettes, TV, voiture dont Renault, machines à laver, fours micro-ondes ...).

Historique

Au départ, la plateforme a été développée par une start-up américaine, Android Inc. La start-up a été rachetée par Google en août 2005.

En 2007, Google crée un consortium nommé OHA (Open Handset Alliance) réunissant une trentaine d'entreprises (opérateurs mobiles, constructeurs , industriels, éditeurs de logiciels …). Le but est de favoriser l'innovation sur les appareils mobiles à l'aide d'une plateforme ouverte et complète. Ce consortium est à l'origine de la sortie du 1er SDK (Software Development Kit) en 2007.

En 2018, l'OHA regroupe plus de 80 sociétés membres.

2) Les alternatives à Android

Les principales sont :

- ✓ iOS (iPhone OS) : les applications sont développées dans les langages Objective-C et Swift. Ces langages sont très différents de Java. De même, globalement il y a de grandes différences entre les systèmes iOS et Android.
- ✓ Windows Phone (successeur de Windows Mobile en 2010) proposé par Microsoft.

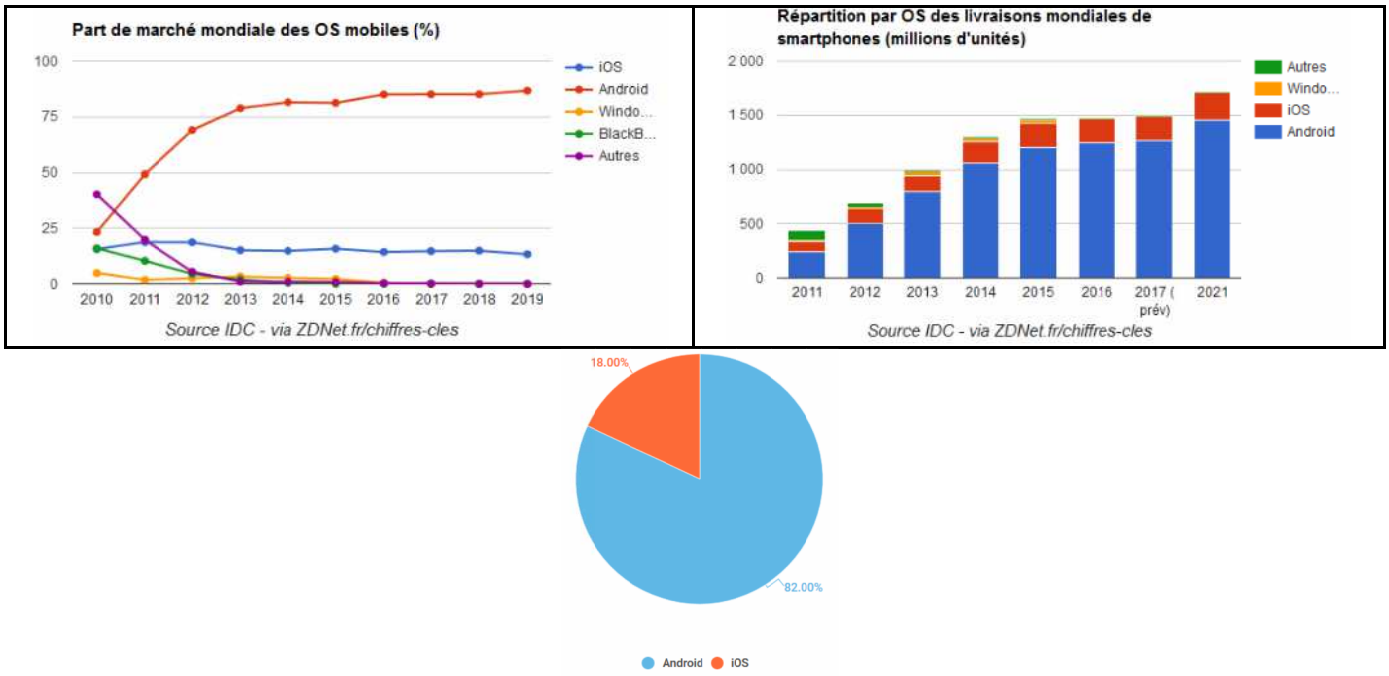
Windows Phone et l'iPhone d'Apple fournissent des environnements de développement riches et simplifiés pour les applications mobiles. Cependant, contrairement à Android, ils sont bâtis sur des

systèmes propriétaires qui, dans certains cas, donnent la priorité aux applications natives (celles pré-installées) au détriment de celles développées par des tiers, limitent la communication entre applications et les données natives du téléphone, et restreignent la distribution des applications tierces.

3) Pourquoi développer sur la plateforme Android ?

D'une manière générale, les appareils mobiles sont le nouveau moyen d'accès à Internet. De plus, le marché des applications mobiles représente un excellent moyen de commercialiser des produits et services. Il est important pour les entreprises d'être à la pointe de l'innovation pour faire face aux concurrents.

Le système Android est installé sur un nombre croissant d'appareils mobiles. Il est très largement le plus utilisé avec une part de marché de 80 % en 2019.



Part de marché des OS mobiles en France (3ème trimestre 2019)

Autres avantages :

- ✓ Les développeurs mettent à profit leurs compétences Java. La documentation disponible sur Internet est importante.
- ✓ La publication des applications est simplifiée.
- ✓ Sous Android, toutes les applications qu'elles soient natives ou pas, ont les mêmes droits. Les possibilités de développement sont donc très grandes. On peut même par exemple, remplacer une application native par sa propre application.

4) La publication des applications

Elle peut se faire sur Google Play Store ou d'autres marchés (magasins d'applications).

Google Play Store était connu auparavant sous le nom d'Android Market (la boutique en ligne de Google). Quelques chiffres :

- ✓ En 2012, il y avait 1 milliard de téléchargements par mois.
- ✓ En 2017 : 2,8 millions d'applications disponibles.
- ✓ En 2016 : les revenus de Google Play sont estimés par les experts à 7.8 billions dollars US.

Les applications proposées sont soit gratuites, soit payantes.

La publication des applications est facile : il faut générer un fichier APK que l'on dépose ensuite sur Google Play. Elle n'est pas soumise à une validation préalable. En fait, l'application sera évaluée par les utilisateurs. Il y a aussi un outil automatique qui analyse l'application et qui est destiné à détecter les applications malveillantes. Cet outil n'est pas fiable à 100 %, environ la moitié des malwares seraient détectés.

Pour publier une application, les droits d'entrée sont faibles. En fait, il faut s'acquitter d'un droit de 25 dollars, une seule fois pour un nombre illimité d'applications. Le but de ce paiement est aussi pour Google de s'assurer que le déposant est bien une personne physique et pas un virus.

Si l'application est payante, Google prélèvera 30 % de son prix, ainsi que sur tout produit ou service vendu via cette application

Les autres marchés sont, par exemple : Amazon AppStore, Samsung Apps, GetJar … Il existe plus de 20 marchés de ce type. En principe, pour autoriser l'installation d'une application venant de l'un de ces marchés, sur son téléphone, il y a un paramètre « Source inconnues » à activer.

5) Les contraintes à prendre en compte dans les applications Android

Tous les appareils mobiles présentent des limites :

- ✓ processeurs moins performants,
- ✓ mémoire restreinte,
- ✓ autonomie très faible.
- ✓ la taille de l'écran est réduite (en général)

Au niveau système, Android (tout comme les autres systèmes) apporte ses contraintes spécifiques :

- ✓ plusieurs versions du système Android coexistent,
- ✓ une grande variété d'appareils mobiles doit être considérée avec plusieurs résolutions d'écran, des fonctionnalités diverses (GPS, accéléromètre, appareil photo…). Ces spécificités sont prises en compte au travers de diverses API (Application Programming Interface)
- ✓ si l'appareil mobile est un smartphone, il reçoit des SMS et des appels qui sont des tâches prioritaires

6) Les versions de la plateforme Android

De multiples versions de la plateforme Android coexistent sur les différents appareils mobiles. Conjointement plusieurs versions d'API existent également.

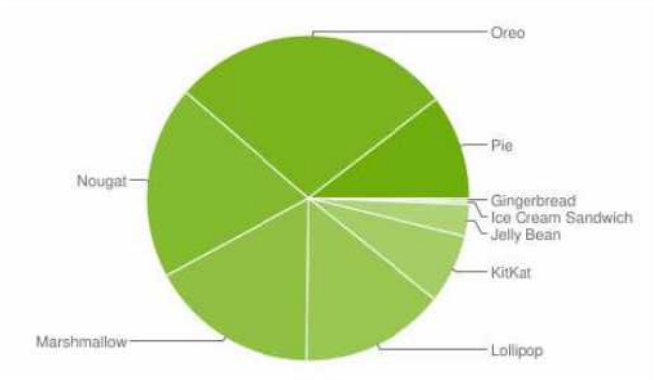
Nom de la plateforme	Version	Niveau d'API	Date
Pas de nom	1.0 et 1.1	API level 1.2	Septembre 2008 – Février 2009
Cupcake	1.5	3	Avril 2009
Donut	1.6	4	Septembre 2009
Eclair	2.0 et 2.1	7	Octobre 2009
Froyo	2.2	8	Mai 2010
Gingerbread	2.3.*	9 et 10	Décembre 2010
Honeycomb	3.0 - 3.2	11 - 13	Février – Juillet 2011
Ice Cream Sandwich	4.0.*	14 - 15	Octobre 2011
Jelly Bean	4.1 – 4.2.*	16 - 17	Juin – Novembre 2012
KitKat	4.4.*	19 - 20	Novembre 2013
Lollipop	5.0	21 - 22	Novembre 2014 - Mars 2015
Marshmallow	6.0	23	Août 2015
Nougat	7.0	24 - 25	Août 2016
Oreo	8.0	26 - 27	Août 2017
Pie	9.0	28	Août 2018
Android 10	10.0	29	Septembre 2019

Quelques apports de quelques versions :

- ✓ **GingerBread** : support du NFC (Near Field Communication ou communication en champ proche), plusieurs caméras sur un même terminal
- ✓ ***Honeycomb*** est une version spécifique pour les tablettes (le but était de minimiser les risques).
- ✓ La version suivante, ***Ice Cream Sandwich***, a permis la fusion de la version pour tablette et de celles pour téléphones (toutefois, certaines tablettes fonctionnent avec 2.3 par exemple, car les constructeurs ont fait le forcing).
- ✓ ***Lollipop*** : changement au niveau de la machine virtuelle : Dalvik est remplacée par ART (Android Runtime)
- ✓ ***Nougat*** : multi-fenêtrage, 2 applications peuvent être ouvertes simultanément et visibles à l'écran
- ✓ ***Oreo :*** une amélioration de fond avec la gestion du multi-tâche amélioré (pour économiser de la batterie), améliorations diverses pour les notifications, *picture in picture* …
- ✓ ***Pie :*** amélioration de l'interface (menu réglage en couleurs, nouveau menu pour régler le son en vertical, transitions animées plus fluides …), adaptation native aux encoches, amélioration de la sécurité (interdiction d'utiliser le micro ou la caméra à l'insu de l'utilisateur, utilisation du protocole HTTPS, sauvegardes chiffrées, restrictions pour le partage d'informations entre les applications …), géolocalisation même au sien d'un bâtiment grâce à la prise en charge d'un nouveau protocole WiFi et à la triangulation, intelligence artificielle plus présente ….
- ✓ ***Android 10 :*** prise en charge native des *smartphones* pliables, *live caption* (des sous-titres peuvent s'ajouter automatiquement), menu de partage plus simple, des paramètres systèmes accessibles directement dans une application, mode sombre natif, intégration de *smart Reply* pour les notifications (propositions de réponse à des messages), renforcement sécurité et vie privée …

Sur le site dédié au développement d'applications Android ***developer.android.com***, on trouve des statistiques pour voir l'évolution de la répartition entre les versions (*dashboards*). Tout développeur doit se tenir informé de ces évolutions.

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.3%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.3%
4.1.x	Jelly Bean	16	1.2%
4.2.x		17	1.5%
4.3		18	0.5%
4.4	KitKat	19	6.9%
5.0	Lollipop	21	3.0%
5.1		22	11.5%
6.0	Marshmallow	23	16.9%
7.0	Nougat	24	11.4%
7.1		25	7.8%
8.0	Oreo	26	12.9%
8.1		27	15.4%
9	Pie	28	10.4%



Proportion de terminaux selon la version d'Android (novembre 2019)

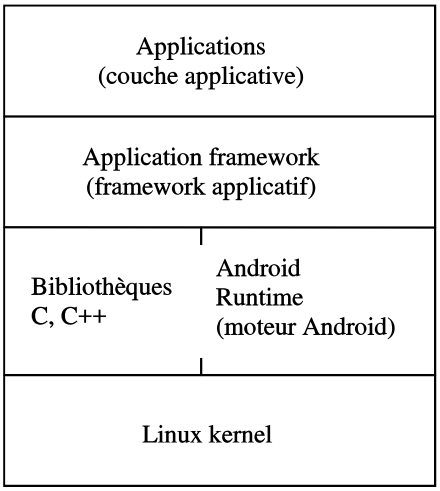
Plusieurs constructeurs, ou marques, proposent des terminaux fonctionnant avec le système Android. A chaque nouvelle version, les constructeurs fournissent plus ou moins rapidement des terminaux avec la cette nouvelle version.

Les programmeurs doivent jongler avec une grande diversité de versions. C'est l'une des caractéristiques d'Android (souvent critiquée par les concurrents). Elle se nomme **fragmentation**.

Bien sûr, il y a une compatibilité ascendante entre les versions : une application développée pour KitKat fonctionnera aussi avec Nougat par exemple. De plus, les nouveautés de chacune des versions ne sont pas indispensables à toutes les applications développées.

Pour couvrir un plus grand nombre d'utilisateurs potentiels, il est bien sûr possible de découper une application en plusieurs niveaux. Une version de base fonctionnerait sur les anciennes versions d'Android, mais ne contiendrait peut-être pas toutes les fonctionnalités, ou l'aspect graphique ne serait pas complètement abouti. Une version complète avec une interface graphique améliorée serait disponible pour la dernière version d'Android.

7) Architecture Android



□ Le **noyau Linux** fait l'interface avec le matériel sous-jacent, ou autrement dit sert de pont entre le matériel et le logiciel. Il gère les processus, la mémoire, les droits des utilisateurs et la couche matérielle. Il faut bien noter qu'il s'agit ici seulement du noyau Linux et pas de la totalité du système Linux. Le code des fonctionnalités (comme par exemple le contrôle de la puce Wifi) est spécifique au matériel. Ce noyau contient aussi les pilotes des dispositifs matériels (USB, affichage, bluetooth …).

La version de ce noyau est adaptée pour les appareils mobiles, avec une gestion avancée de la batterie, et une gestion particulière de la mémoire, par exemple. Ce noyau Linux est mis à jour régulièrement selon les versions du système (premières versions avec 2.6.x, ensuite 3.4.x, actuellement 4.9)

□ Dans la partie **Android Runtime** (Environnement d'exécution Android, ou moteur d'exécution c'est-à-dire un programme qui permet l'exécution d'autres programmes), on trouve

- Des bibliothèques de base du Java, et certaines spécifiques à Android
- la machine virtuelle Dalvik qui fonctionne conjointement avec les bibliothèques. Elle est optimisée pour une consommation mémoire minimale et garantit qu'un appareil peut exécuter efficacement plusieurs instances. Elle est fortement couplée au noyau Linux sous-jacent (gestion de la mémoire bas niveau et des threads, la sécurité)

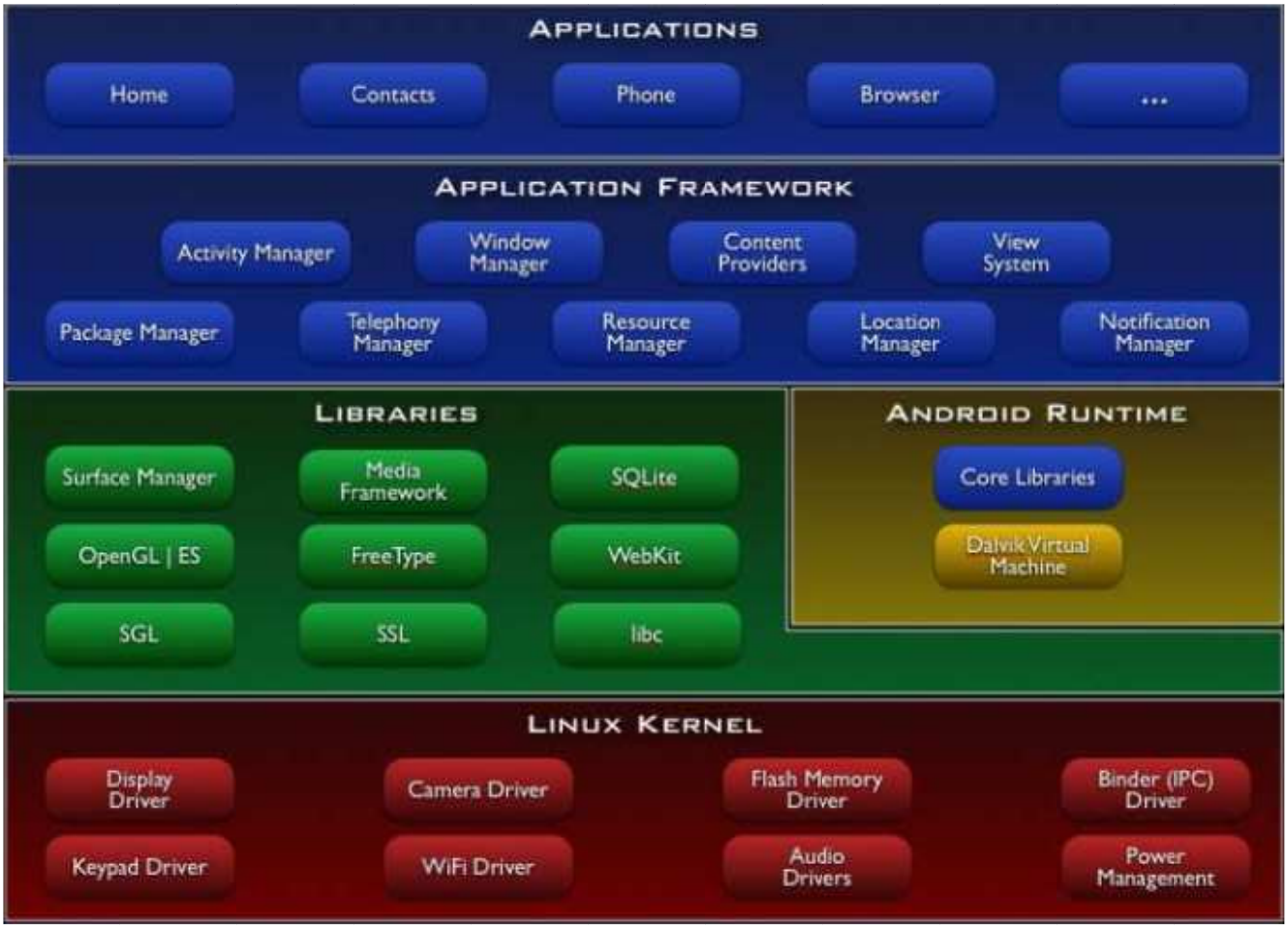
□ Les **bibliothèques C, C++** offrent des accès bas niveau aux applications : bibliothèques systèmes (accès aux fonctions de base du système), bibliothèques de manipulation des médias (audio, vidéo …), moteur du navigateur WebKit, moteur du gestionnaire de bases de données SQLite, OpenGL (pour les graphiques 2D ou 3D) … Ses bibliothèques sont open-source.

□ Dans le **framework applicatif,** il y a des bibliothèques Java qui appellent les bibliothèques C et C++. Elles permettent aux applications d'interagir avec le système Android, notamment grâce à des classes jouant le rôle de gestionnaire des différentes ressources. Exemples de bibliothèques : Activity Manager, View System, Window Manager (gestionnaire de fenêtres), Location Manager (gestionnaire de géolocalisation), Content Providers (gestionnaires de contenu), gestion de la téléphonie, gestion des les capteurs.

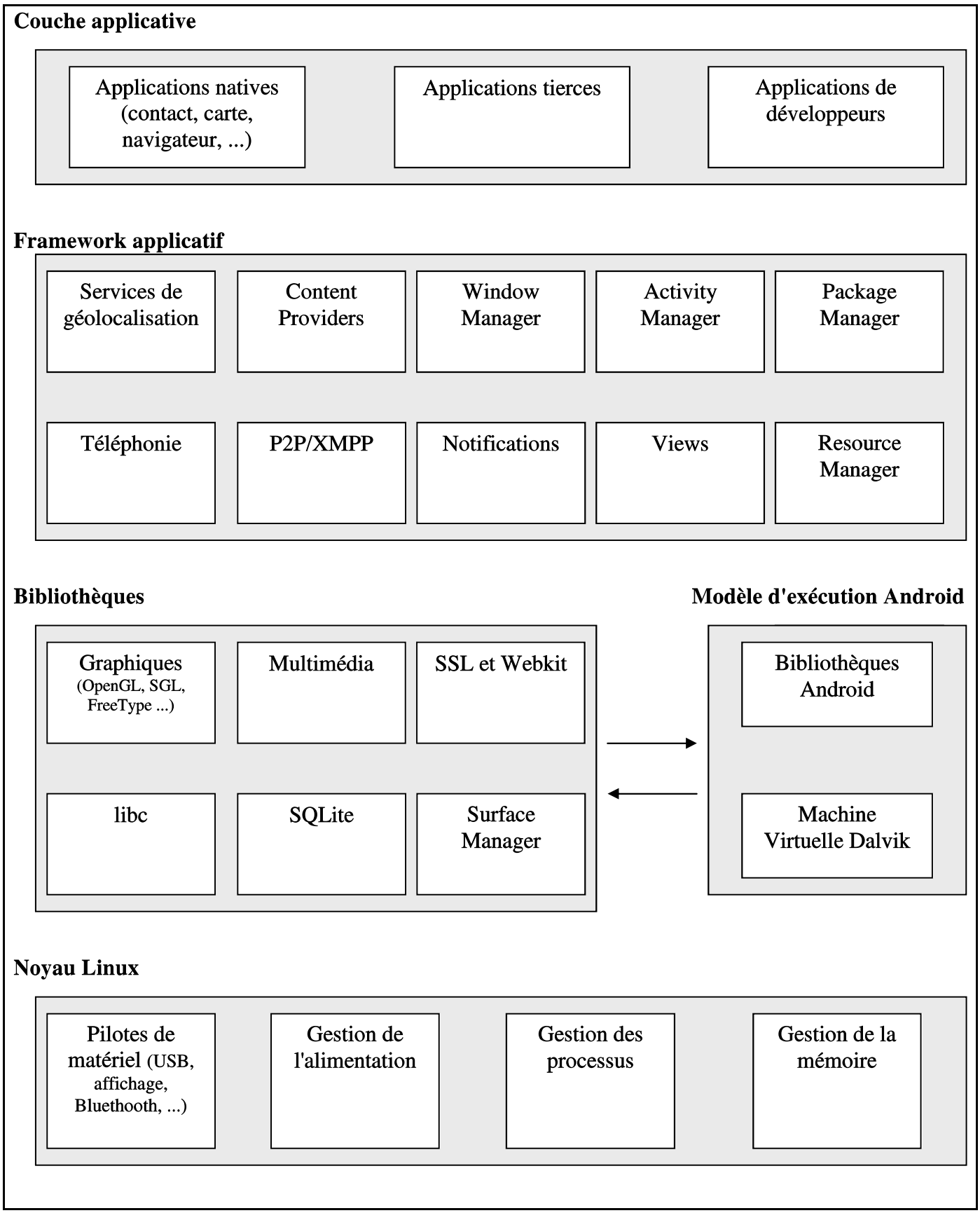
□ La dernière couche est celle des **applications**. Il s'agit ici des applications standards préinstallées sur les appareils : gestion des e-mails, appareil photo, gestion des contacts … et aussi des applications développées par des tiers.

Remarque : il est possible de développer des applications en C ou C++ pour Android. Par exemple, si on souhaite développer une application de traitement d'images, donc gourmande en ressources, on pourrait coder en C++ pour un gain d'efficacité. On utilise alors le NDK (Native Development Kit). Autres exemples où des performances élevées sont exigées : jeu, analyse de signal ...

Schéma d'architecture extrait du site developer.android.com :

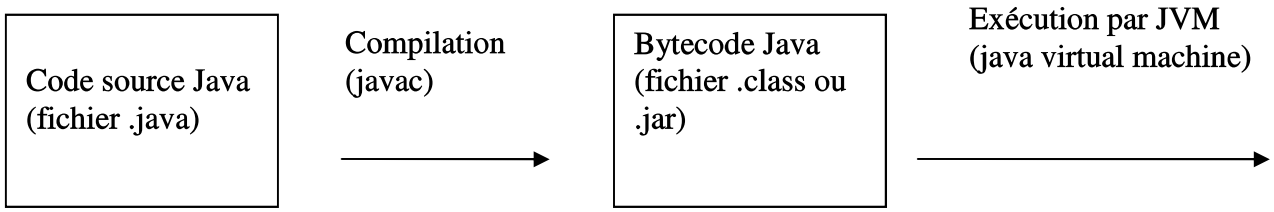


Autre version du schéma d'architecture:



8) La machine virtuelle Dalvik et Android Runtime (ART)

Rappel : comment générer un programme exécutable en Java standard ?



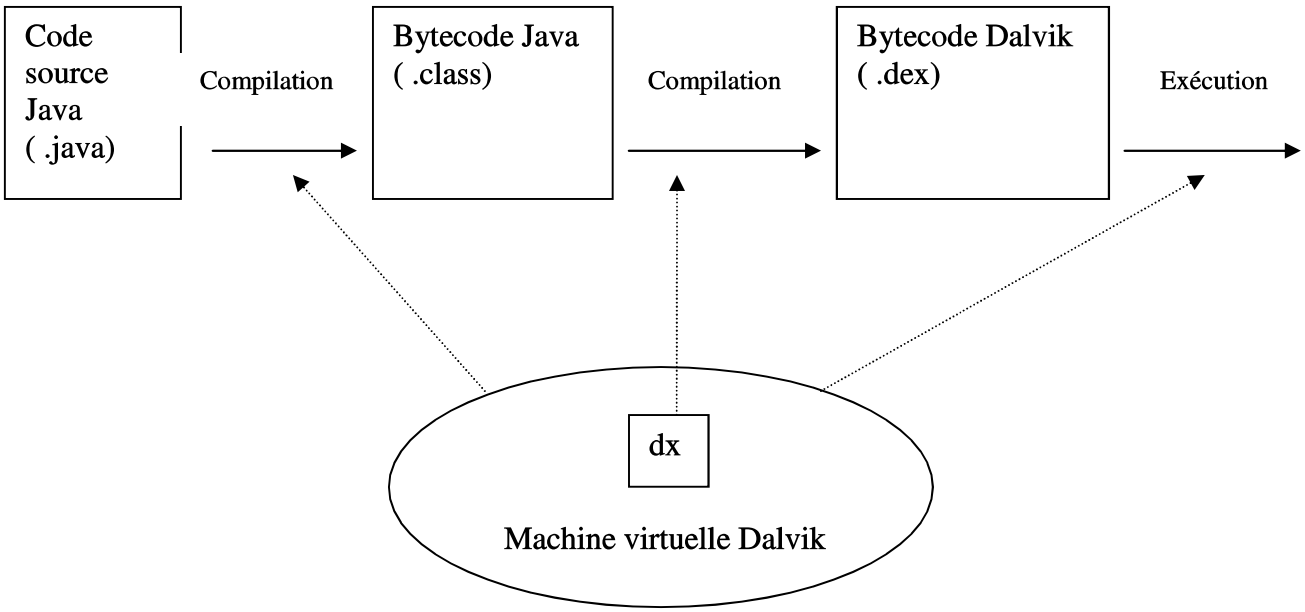
Avec Android :

La version de Java qui permet le développement d'applications mobiles est une version réduite dépourvue de certaines fonctionnalités inutiles. Par exemple, la bibliothèque Swing n'est pas intégrée.

Avant la version 5 d'Android (jusqu'à KitKat) : La machine virtuelle Dalvik :

Une machine virtuelle a été spécifiquement conçue pour les systèmes embarqués, elle se nomme Dalvik. Elle est optimisée pour gérer les ressources physiques du système : optimisation de l'occupation mémoire d'une application, de l'utilisation de la batterie …

Chaque programme qui s'exécute le fait sur sa propre instance la machine virtuelle Dalvik.



Dx est un programme dont le rôle est de traduire du bytecode Java en bytecode Dalvik.

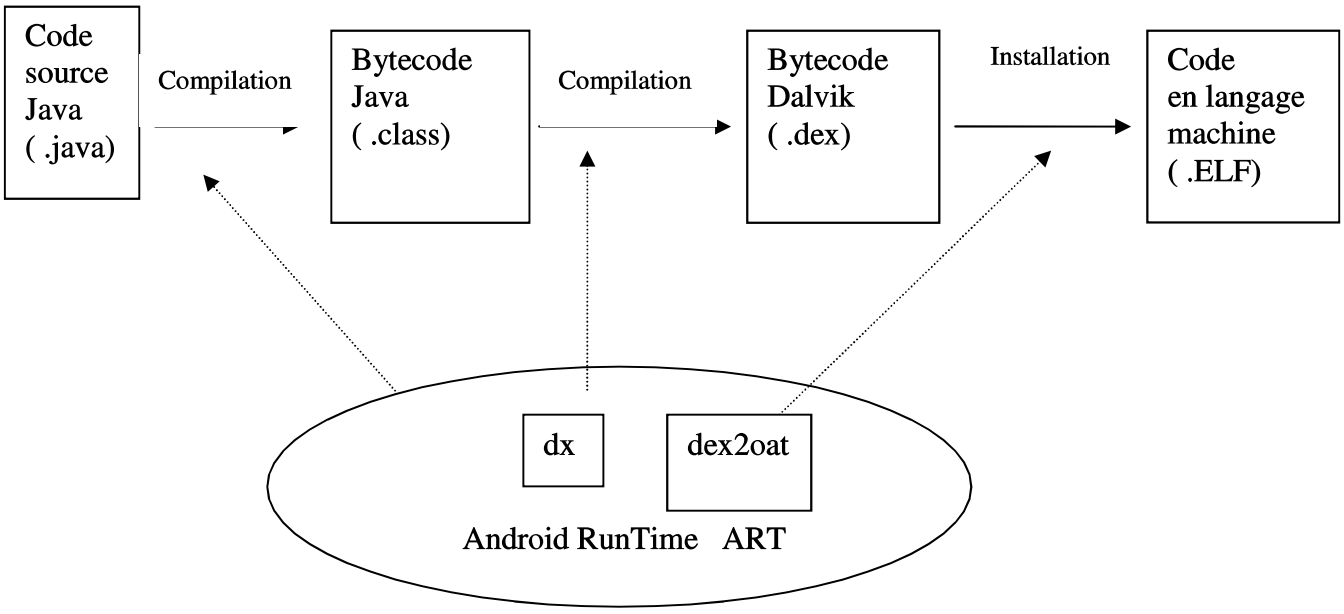
Le code final d'une application est empaqueté dans un fichier APK (Android Package). Ce code est fabriqué par l'outil AAPT (Android Asset Packaging Tool).

Depuis la version 5 d'Android (à partir de Lollipop) : Dalvik a été remplacée par ART :

ART signifie Android RunTime. Le *bytecode* n'est plus traduit en langage machine "à la volée" au moment de l'exécution du programme (ou just-in-time).

Mais il est compilé, traduit en langage machine au moment de son installation sur l'appareil. On parle de *ahead-of-time* (AOT) compilation (ou compilation préalable ou anticipée).

Pour des raisons de compatibilité, ART utilise le même type de fichier que Dalvik, à savoir des fichiers .dex, et produit des fichiers exécutables dans le format .ELF (Executable and Linkable Format).



Avantages d'ART

- ✓ les applications seront plus fluides, puisqu'il n'y aura plus, au moment de leur exécution, la phase de traduction à la volée
- ✓ en conséquence, et c'est l'objectif recherché, les ressources de la batterie sont moins consommées

Avantage de Dalvik qui sont perdus, inconvénients d'ART

- ✓ le code en langage machine occupe plus de place en mémoire, mais c'est peu génant car les capacités des appareils mobiles augmentent
- ✓ les temps d'installation d'une application sont plus importants (surtout génant pour les développeurs en phase de tests)

ART a amené aussi des améliorations du *garbage collector*, des facilités pour la mise au point des programmes : les messages d'erreurs sont plus détaillés.

Remarques :

- Il existe une version spéciale de Java destinée au développement mobile, mais pas pour Android : Java ME
- Des téléphones utilisent des machines virtuelles autres que Dalvik, c'est le cas d'un Nokia par exemple.
- Comme Android est open-source, plusieurs fabricants de téléphones ou des opérateurs ont personnalisé l'interface utilisateur et les applications fournies avec chaque appareil. Ainsi l'interface d'un Samsung n'est pas la même que celle d'un Nexus.
- On peut développer en C et C++. Ces applications sont alors directement exécutées par le système d'exploitation Linux.

9) Conclusion

Java est le principal moyen de construire des applications Android. Mais ce n'est pas le seul. Outre les langages C et C++, on peut aussi utiliser Perl, Python, Scala, Clojure, HTML 5. Les performances sont en général moins bonnes, sauf bien sûr avec C ou C++.

Il existe des frameworks d'applications qui permettent de développer des applications à partir d'une base de code unique et de les déployer sur différentes plates-formes système (Android, Windows, iOS ...). Ce point est détaillé en annexe.

Annexe – 4 manières d'aborder le développement mobile

La diversité des plateformes mobiles (Android, iOS, Windows) est un inconvénient pour les entreprises du numérique. En effet, il est parfois nécessaire de développer une application pour deux plateformes, voire trois. Ces multiples développements ont l'inconvénient d'augmenter les coûts, y compris pendant la phase de maintenance.

Pour créer une application iOS, on utilise l'environnement XCode, et les langages Objective-C ou Swift, ainsi que les Storyboard pour définir des interfaces graphiques, par exemple. Pour une application Android, on fait appel à Android Studio, et aux langages Java ou Kotlin et XML. Pour Windows, on développe sous Visual Studio avec les langages C# ou VB.NET, par exemple.

Toutefois, il n'est pas toujours nécessaire de produire une application pour plusieurs plateformes. Par exemple, s'il s'agit de développer une application destinée aux commerciaux de l'entreprise et s'ils sont tous équipés d'un téléphone professionnel fonctionnant sous Android, une seule plateforme sera visée.

Il existe 4 manières d'aborder le développement mobile et de prendre en considération les diverses plateformes :

- ✓ le développement natif
- ✓ le développement web
- ✓ le développement hybride
- ✓ le développement multi-plateforme natif

A) Le **développement natif** utilise les langages, les API et les outils proposés par les éditeurs de la plateforme ciblée. Donc quand on effectue un développement, il concerne une plateforme précise. Si on souhaite que notre application soit utilisable sur deux plateformes par exemple, il faudra la coder deux fois.

Les avantages sont les suivants :

- ✓ en résumé une application plus performante et de meilleure qualité
- ✓ on peut exploiter les spécificités du système. L'application native a accès à toutes les spécificités du terminal mobile
- ✓ on obtient une interface homogène avec les applications natives du système,
- ✓ les applications sont plus réactives ce qui est un point très important pour les applications mobiles,
- ✓ on peut utiliser toutes les APIs et ressources de la plateforme …

Les inconvénients sont bien sûr le coût pour l'entreprise (multiplication des développements et maintenances) et les délais augmentés. Il faut disposer des compétences techniques, ce qui implique de former les programmeurs aux différentes plateformes donc une augmentation du coût également.

B) Le **développement web** : on développe une application purement web. Le principal avantage est que l'on obtient immédiatement une application accessible depuis toutes les plateformes (gain de temps et réduction du coût). De plus, les technologies utilisées sont généralement bien connues des programmeurs.

Cependant, les applications web nécessitent obligatoirement une connexion internet et elles seront moins fluides et réactives que les applications natives. Elles ne pourront pas exploiter toutes les spécificités du système. Il n'y aura pas de possibilité d'accéder aux capteurs par exemple.

C) Le **développement hybride** fait appel aux technologies Web (HTML/CSS/Javascript). Chaque plateforme native fournit un composant permettant d'afficher un navigateur web, nommé *Webview*. Le composant *Webview* est hébergé et affiché par une application native appelée « conteneur » (application très réduite). L'application elle-même s'exécute au sein de la *Webview*. Le conteneur peut éventuellement faire appel aux APIs natives.

Les avantages sont un coût de développement réduit (développement de l'interface et du code métier effectué une seule fois pour les différentes plateformes), un développement rapide, des compétences plus classiques exigées pour les développeurs.

Les inconvénients sont des applications moins performantes, le style graphique n'est pas celui des applications natives, on n'a pas d'accès aux spécificités du système, pas d'accès possible à la carte graphique par exemple.

Il existe plusieurs *frameworks* permettant le développement hybride. Le principal est Cordova (son ancien nom est PhoneGap, encore utilisé). Il est *opensource*. De nombreux IDE le prennent en charge, comme par exemple Visual Studio 2017. Cordova repose sur une architecture modulaire de *plugins*. Chaque *plugin* sert de pont entre Javascript et une implémentation native spécifique à une plateforme. Pour qu'un *plugin* fonctionne sur 3 plateformes, il faut qu'il existe 3 implémentations natives de celui-ci. Cependant tous les *plugins* ne sont pas disponibles en version 3 plateformes, ce qui est limitatif.

Notons que Cordova n'inclut pas d'éléments pour réaliser une interface graphique ou structurer le code d'une application. Mais on peut lui adjoindre *Ionic* et *Angular*, par exemple, pour répondre à ces deux besoins. Un autre *framework* permettant ce type de développement est *Appcelerator Titanium.*

D) Avec **l'approche multi-plateforme natif**, l'idée est d'utiliser un seul et même langage de programmation pour toutes les plateformes, tout en utilisant les API natives à chacune des plateformes. On code une seule application qui pourra fonctionner sur plusieurs plateformes mobiles. Une grande proportion de code est commune aux diverses plateformes.

Le framework Xamarin, racheté par Microsoft en 2016, est la solution la plus populaire. Le développement est réalisé en C#. L'IDE utilsé est Visual Studio complété par Xamarin, ou bien Xamarin Studio.

Les performances des applications obtenues sont quasi-identiques aux applications natives, l'interface graphique est riche et intégrée au système, on peut utiliser 100 % des API natives de chaque plateforme. La plus grande partie du code est commune aux différentes plateformes, une petite partie reste à coder de manière spécifique pour chacune des plateformes :

- ✓ Soit de 70% à 85 % si on utilise l'approche classique (on utilise les fichiers graphiques natifs)
- ✓ Soit de 90% à 98% si on utilise Xamarin Forms (on utilise une API graphique indépendante qui sera ensuite transposée nativement)

Notons qu'il existe des approches alternatives à Xamarin. Citons par exemple, Codename One, dans laquelle le développement se fait en Java. Il s'agit d'un plugin que l'on peut ajouter à plusieurs IDE, dont Eclipse. La compilation du code se fait dans le Cloud. Le code généré, à partir d'un code unique, est natif : pour iOS, on obtient du C, pour Windows du C#.

Autres alternatives : Qt (on code en C++), Windev Mobile (on code dans un langage spécifique, le W-langage. Il s'agit d'une solution payante mais souvent utilisée en entreprise), C++ Builder.

Les bases du développement Android

I) L'outil de développement Android Studio

Au niveau des outils de développement, l'environnement qui a été privilégié pendant longtemps était Eclipse auquel un plugin était ajouté : Eclipse ADT (Android Developer Tools). L'outil recommandé actuellement est Android Studio.

Android Studio est édité par Google (depuis 2014 pour la première version) et est disponible sous Windows, Mac OS et Linux. Il s'appuie sur le JDK (Java Development Kit). Il est basé sur l'environnement de développement **IntelliJ IDEA** de la société JetBrains

Android Studio contient plusieurs outils. Citons par exemple :

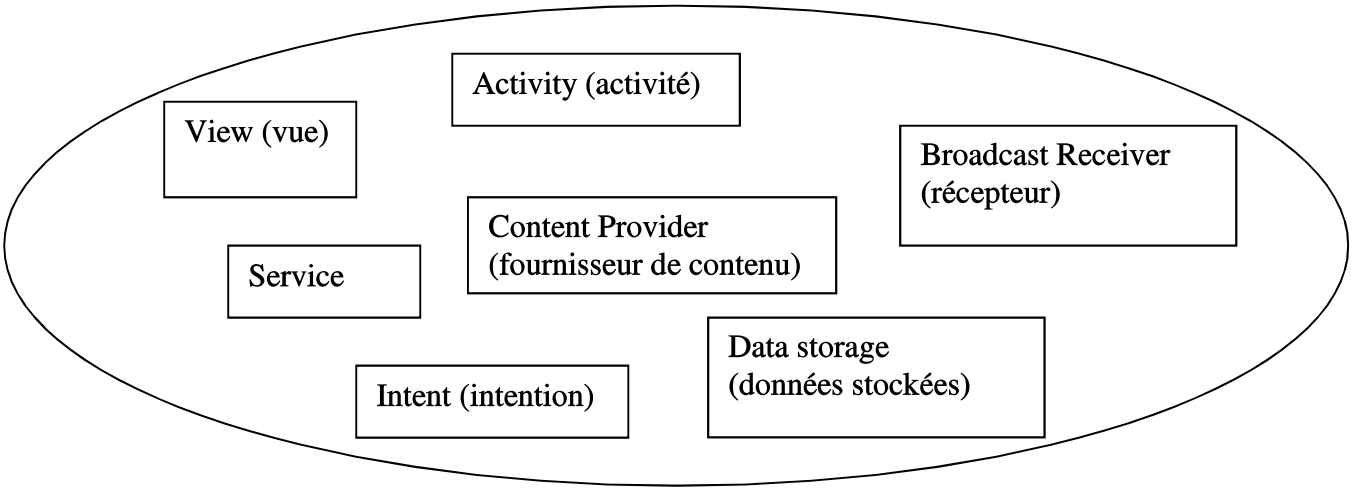
- ✓ Un moteur de production (ou *build system*) nommé **Gradle**
- ✓ Un outil pour construire visuellement des interfaces graphiques
- ✓ Des modèles d'applications correspondant aux principales structures de projets Android

Un autre outil important est le **SDK and AVD Manager** qui permet de télécharger les composants spécifiques aux versions de la plateforme Android sur lesquelles nous souhaitons faire tourner les applications développées.

Le kit de développement Android contient un émulateur appelé **AVD** (Android Virtual Device). Il permet de paramétrer plusieurs configurations : différentes versions d'API et différentes résolutions d'écran par exemple. Les simulateurs incluent des aspects matériels comme par exemple la taille de la mémoire ou l'appareil photo. Ces appareils virtuels sont gérés par le gestionnaire d'appareils virtuels Android : **AVD Manager** (**Android Virtual Device Manager**).

II) Les composants d'une application Android

Une application Android est un assemblage de composants faiblement couplés.



Tous ces composants seront assemblés dans un fichier ***.apk*** (fichier zip, Android package) pour constituer l'application à installer.

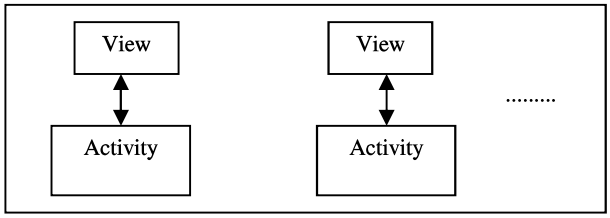
L'application elle-même est décrite dans un fichier XML : *AndroidManifest.xml*. On énumère et on configure dans ce fichier les composants clés de l'application. On y décrit les caractéristiques de l'application, les bibliothèques externes, les permissions requises ....

Les grands principes du développement sous Android

□ L'utilisateur interagit avec une application Android par le biais des **activités** (**activity**). Une application est constituée d'une ou de plusieurs activités. Quand l'utilisateur veut utiliser une application, il clique sur une icône du bureau, ce qui permet de lancer en fait l'activité principale de cette application.

A chaque activité est associé un écran de présentation qui affiche des informations et permet à l'utilisateur de saisir des données. Cet écran se nomme une **vue** (**view**).

Une application :



A un instant *t*, une seule activité est en cours d'exécution. Les activités lancent d'autres activités en envoyant des intentions (*intent*) au framework.

□ Les **intentions** sont des messages envoyés au framework applicatif dans le but de déclencher des actions, comme par exemple lancer une activité. Il existe deux types d'intentions :

- Les **intentions explicites** dans lesquelles ont décrit exactement le composant destinataire. On demande l'activation d'une classe dont on fournit le nom.
- Les **intentions implicites** utilisées pour demander l'exécution d'une action générale. Par exemple, on peut demander à déclencher l'appareil photo. C'est le framework applicatif qui décide alors comment interpréter l'action demandée. Si par exemple, plusieurs appareils photos sont présents, il affichera une fenêtre pop-up pour demander à l'utilisateur de choisir.

Le framework applicatif joue donc le rôle d'un facteur, il distribue les messages c'est-à-dire les intentions.

Le terminal émet aussi des messages systèmes sous la forme d'intentions. Ces messages servent à prévenir les applications de la survenue de différents événements : insertion d'une carte SD, réception d'un SMS par exemple.

□ Les **services** exécutent les opérations longues en tâches de fond (arrière plan). Le but est de ne pas bloquer le terminal sur un tâche longue. Ils n'ont pas d'interface utilisateur. Par exemple : accéder à un fichier, échanger avec des services Web, diffuser de la musique … Il existe 2 types de services : ceux qui s'exécute dans le même processus que l'application (**LocalService**) et ceux qui s'exécutent dans des processus indépendants de l'application (**RemoteService**)

□ Un **fournisseur de contenus** (**content provider**) peut être vu comme un protocole standardisé pour accéder à des données (stockées sous diverses formes). Dit autrement, ce sont des abstractions pour accéder à des données. Ils sont notamment utiles pour partager des données entre applications. Les applications natives utilisent aussi ce mécanisme. Par exemple, le gestionnaire de contacts organise ses données sous la forme d'un fournisseur de contenu. Dans notre propre application, nous pourrons donc accéder à celui-ci et donc consulter et/ou agir sur les contacts.

□ Les **récepteurs** de diffusion (**broadcast receivers**) : ils permettent de capter les messages diffusés via les intentions. Si on crée un *broadcast receiver* dans une application, celle-ci pourra intercepter les intentions répondant à des critères que l'on aura spécifiés grâce à un filtre. L'application peut alors jouer le rôle d'un gestionnaire d'événements.

□ **Autres composants (à un autre niveau)**

- ✓ Les **widgets** sont les composants visuels d'une application.
- ✓ Les **notifications** permettent d'envoyer un signal aux utilisateurs sans dérober le focus ni interrompre l'activité en cours. Elles permettent d'attirer l'attention de l'utilisateur. L'idée est la même que celle utilisée par les applications natives pour informer l'utilisateur qu'un SMS a été reçu, ou qu'un appel est arrivé.

III) Structure d'un projet Android

Selon l'environnement de développement utilisé, les dossiers et sous-dossiers d'un projet peuvent se situer à des emplacements différents. Les dossiers et fichiers principaux sont les suivants :

Fichier `AndroidManifest.xml`	Il décrit l'application à construire et ses composants (activités, services, permissions, fournisseurs de contenus, intentions auxquelles l'application doit répondre ...). Un composant ne peut être exécuté que s'il est déclaré dans le fichier manifeste.
`bin`	Contient l'application compilée
`libs`	Contient les fichiers .jar extérieurs nécessaires à l'application
`res`	Contient les ressources (icônes, description de l'interface graphique ...)
`src`	Contient le code source de l'application
`assets`	Contient les autres fichiers statiques fournis avec l'application pour son déploiement sur le terminal
`gen`	Contient le code source produit par les outils de compilation Android (`gen` = generated)

Répertoire **res** Ce répertoire contient les ressources de l'application (voir le chapitre suivant)

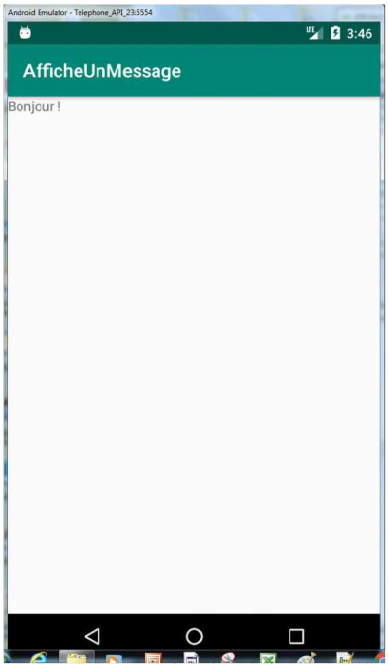
`res/drawable/`	Les images sous la forme de fichiers png, jpeg, ou gif ou bien de fichiers XML pour décrire les dessins simples (cercles, carrés ...)
`res/layout/`	Description XML de l'interface graphique
`res/menu/`	Description XML des menus
`res/raw/`	Fichiers généraux : clip audio, fichier CSV ...
`res/values/`	Définitions des constantes : chaînes de caractères, dimensions ...
`res/xml/`	Les autres fichiers XML

Répertoire **bin**

`bin/classes/`	Les classes Java compilées
`bin/classes.dex`	L'exécutable créé à partir des classes compilées
`bin/application.ap_`	Contient les ressources de l'application sous la forme d'un fichier .zip (*application* est le nom de l'application)
`bin/application-*.apk`	L'application Android complète (apk = Android package) sous la forme d'un fichier .zip

IV) Un premier exemple

On souhaite développer une application qui, une fois lancée, affiche sur l'écran du terminal "Bonjour !". Cette application est constituée d'une seule activité (et donc d'une seule vue).



Il faudra bien sûr créer un projet Android (voir TP). Cette opération va permettre de générer plusieurs fichiers, et le programmeur devra modifier généralement 3 d'entre eux :

- le fichier Java contenant le code de l'activité principale (la seule de l'application)
- le fichier XML contenant la vue associée à l'activité
- le fichier de ressources `strings.xml` qui contient la définition des constantes chaînes de caractères

Le fichier XML pour définir les constantes chaînes de caractères (*strings.xml*)

```
<!-- Chaînes de caractères utilisées dans l'application nommée
     AfficheUnMessage
     fichier strings.xml                12/19 -->
<resources>

    <!-- chaîne générée automatiquement : contient le nom de l'application -->
    <string name="app_name">AfficheUnMessage</string>

    <!-- Texte affiché sur la vue de l'activité -->
    <string name="messageAAfficher">Bonjour !</string>
</resources>
```

Ce fichier définit deux constantes :

- l'une pour le nom de l'application. C'est le nom qui est affiché sur le terminal.
- l'autre pour la chaîne contenant le texte à afficher.

Le fichier XML pour définir la vue (fichier *activity_main.xml*)

Techniquement, il est possible de définir les composants de l'interface graphique dans le fichier Java décrivant l'activité. Mais on préfère utiliser un fichier de positionnement (*layout*) codé en XML (codage plus simple avec XML, on peut utiliser un assistant, code plus lisible d'autant plus que l'on sépare la vue des traitements).

Nous serons toutefois parfois amenés à modifier des composants, ou à en créer de nouveaux de manière dynamique. Dans ce cas-là, il faudra coder ces évolutions dans le fichier Java. En effet, dans le fichier XML ne peuvent figurer que des éléments statiques.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fichier xml décrivant la vue associée à l'activité de l'application
     nommée AfficheUnMessage
     fichier activity_main.xml              12/19 -->
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <!-- Widget permettant d'afficher le message bonjour -->
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/messageAAfficher"/>

</LinearLayout>
```

Ce fichier contient d'abord une balise `<LinearLayout>`. On définit ainsi quel gestionnaire de mise en forme doit être utilisé pour présenter les composants sur la vue. Comme son nom l'indique, `LinearLayout` place les composants de manière linéaire les uns à la suite des autres. Ici l'alignement se fait de manière verticale, comme indiqué par la propriété `android:orientation="vertical"`.

De plus, le gestionnaire occupera tout l'espace disponible sur le terminal, aussi bien en largeur qu'en hauteur :

```
android:layout_width="match_parent"
android:layout_height="match_parent"
```

Dans l'élément racine du fichier XML, il faut indiquer l'espace de noms XML d'Android :

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

Tous les autres éléments du fichier XML héritent de cette déclaration. La propriété suivante vient en complément de la première.

```
xmlns:tools="http://schemas.android.com/tools"
```

La propriété : `tools:context=".MainActivity"` doit indiquer le nom de l'activité associée à la vue .

Un seul composant sera présent sur la vue, un composant de type `TextView`, c'est-à-dire une étiquette. Ce composant est décrit dans la balise `<TextView>`. 3 propriétés sont mentionnées :

- la largeur du composant : elle sera celle de son contenu
- la hauteur du composant : également celle du contenu
- le texte présent sur l'étiquette : il s'agit de la constante *messageAAfficher* définie dans un fichier de ressources (le fichier `strings.xml`).

A noter, les composants graphiques sont appelés des **widgets**.

Une classe pour coder une activité

```
/*
 *  Exemple simpe d'application Android
 * fichier MainActivity.java                              12/19
 */
package com.cours.exemple.affichemessage;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;

/**
 * Activité principale de l'application qui affiche un texte sur le terminal
 * Notons que cette classe hérite de AppCompatActivity (et pas de Activity)
 * pour que l'application puisse fonctionner sur les anciennes versions
 * d'Android
 * @author Servières
 * @version 1.0
 */
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

Une activité est codée sous la forme d'une classe qui hérite de la classe prédéfinie **`Activity`**. Elle doit redéfinir (au moins) la méthode **`onCreate`** qui est appelée par le *framework* lorsque l'activité est prête à être lancée. Cette méthode peut être assimilée à l'équivalent d'un constructeur.

L'argument `savedInstanceState` est de type **`Bundle`**. Cet objet contient les données relatives à l'activité. Plus précisément, au premier lancement de l'application, ou lors d'un redémarrage après un arrêt normal, il a la valeur *null*. Par contre, si l'application retrouve le focus après l'avoir perdu, et si le programmeur a bien codé la classe correspondant à l'activité (voir plus loin), l'argument contiendra toutes les données relatives à l'activité.

Dans le corps de cette méthode, il faut obligatoirement :

- appeler la méthode **`onCreate`** de la classe de base **`Activity`**
- associer une vue à l'activité, en appelant la méthode **`setContentView`**. Cette méthode a en paramètre l'identifiant correspondant au fichier XML contenant la vue. Ici le fichier est `activite_principale.xml`, présent dans le répertoire `layout`. Cet identifiant est défini dans la classe auto-générée ***R*** (voir plus loin).

En effet, comme indiqué précédemment les ressources d'un projet Android sont stockées dans le dossier `res` de l'arborescence. ADT (Android Development Tool) interprète ces ressources et les rend accessibles dans le code Java via une classe auto-générée : la classe ***R*** (d'où l'expression `R.layout.activite_principale).`

Dans la classe définissant une activité, si nécessaire, on surchargera également d'autres méthodes provenant de la classe **`Activity.`**

Remarque importante - Lien entre code Java et XML

Si dans le code Java, on souhaite accéder à l'un des *widgets* définis dans le fichier XML, il faut anticiper et dans le fichier XML associer un identificateur, ou identifiant, au *widget*.

```
<TextView
      android:id="@+id/monTextView"
      android:layout_width="wrap_content"
```

La présence du caractère '+' après '@' indique que l'on définit un nouvel identificateur, ici `monTextView`. Le *widget* pourra être accédé dans le code Java en utilisant la méthode `findViewById`.

```
TextView monTexte = (TextView) findViewById(R.id.monTextView);
```

Le fichier XML *AndroidManifest.xml*

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.affichemessage"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="11"
        android:targetSdkVersion="21" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".ActivitePrincipale"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Chaque projet Android inclut un manifeste, stocké dans un fichier XML nommé `AndroidManifest.xml` stocké à la racine de l'arborescence du projet. Il permet de définir la structure du projet, les ressources utilisées, les composants, les pré-requis, les permissions ... En résumé, il contient toutes les informations dont a besoin le système pour installer et exécuter l'application.

Les nœuds du fichier décrivent les composants (activités, services, fournisseurs de contenu, récepteurs de diffusion ...) de l'application, et grâce à des filtres d'intention ou de permissions on précise comment ils interagissent les uns avec les autres, ainsi qu'avec les autres applications.

En particulier, chaque activité qui constitue l'application doit être inscrite dans le fichier AndroidManifest. On indique également au système quelle activité doit être lancée au démarrage de l'application.

Explications de l'exemple

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.affichemessage"
    android:versionCode="1"
    android:versionName="1.0" >
```

L'attribut `package` indique le nom du package contenant l'application.
L'attribut `versionCode` définit la version de l'application, c'est la valeur utilisée en interne.
L'attribut `versionName` indique la version publique affichée aux utilisateurs.

Ensuite, la balise `manisfest` contient des nœuds qui définissent les composants de l'application.

```
<uses-sdk
        android:minSdkVersion="11"
        android:targetSdkVersion="21" />
```

Le nœud `uses-sdk` définit les versions minimale et maximale et cible (éventuellement) qui doivent être disponibles sur un appareil pour que l'application fonctionne correctement. La version est la version pour laquelle l'application a été optimisée.

```
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
```

Un `manifest` ne peut contenir qu'un seul nœud application.

En principe, `allowBackup` sera égal à vrai. S'il est égal à faux, cela signifie que l'application ne pourra pas être restaurée, si elle est mise en attente dans la pile d'exécution par exemple.

On définit ensuite l'icone utilisée pour représenter l'application (par défaut le bonhomme vert, logo Android), le nom de l'application et le thème (il s'agit du style qui s'applique à l'ensemble de l'application). Ces 3 éléments seront présents dans le dossier `res` de l'application, sous la forme de ressources : un fichier image, une chaîne dans `strings.xml`, un style dans le fichier `style.xml`.

```
      <activity
            android:name=".ActivitePrincipale"
            android:label="@string/app_name" >
```

Le caractère point dans le nom de l'application représente le package de l'application. On retrouve au niveau du label, le nom de l'application, puisque notre exemple comporte une seule activité.

```
      <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
```

Il s'agit de décrire ici un filtre d'intentions, ou autrement dit de décrire les intentions qui sont susceptibles de déclencher l'activité. Ce filtre doit contenir au moins une balise `action`.

La balise `action` précise que l'activité va accepter l'action "`android.intent.action.MAIN`". On précise ainsi que cette activité sera considérée comme le point d'entrée de l'application.

Avec la balise `category`, on dit que l'icône de l'activité sera placée sur le lanceur du terminal (écran principal à partir duquel on lance les applications)

Quelques précisions sur la classe *R* (classe générée automatiquement et présente dans le dossier `gen`, ou si Android Studio a été utilisé : `app\build\generated\not_namespaced_r_class_sources\debut\r\` *`nom du package`*).

La classe R ci-dessous est un extrait de la classe R complète. Dans cet extrait, seuls les identifiants spécifiques à l'application programmée sont mentionnés.

```
public final class R {

    public static final class drawable {
        public static final int ic_launcher=0x7f020057;
    }

    public static final class layout {
        public static final int activite_principale=0x7f030018;

    }

    public static final class string {
        public static final int app_name=0x7f0a000d;
        public static final int messageAAfficher=0x7f0a000e;
    }
    .....
```

La structuration de la classe ressemble à celle du répertoire de ressource `res`, mais ce n'est pas une copie conforme de celui-ci.

A partir des ressources, les outils ont généré des classes internes à la classe ***R***. Ces classes internes contiennent des constantes hexadécimales de type ***int***, initialisées avec l'adresse en mémoire de l'entité définie.

Nous pourrons accéder à celles-ci **dans le code Java** en écrivant, par exemple :
- ***`R.layout.activite_principale`*** si nous souhaitons accéder au fichier XML contenant la vue associée à l'activité principale
- ***`R.string.messageAAfficher`*** si nous souhaitons accéder à la chaîne de caractères ayant l'identifiant *messageAAfficher*

Nous pourrons accéder aux mêmes ressources **dans le code XML** en écrivant :
- ***`@layout/activite_principale`***
- ***`@string/messageAAfficher`***

V) Le moteur de production Gradle

Gradle est un moteur de production fonctionnant sur la plateforme Java. Il est intégré à l'IDE Android Studio. C'est lui qui permet de construire les projets contenant les applications. Il est chargé d'organiser la compilation en indiquant quels sont les fichiers sources à compiler, en fonction des modifications faites par le développeur, quelles sont les bibliothèques externes à utiliser, où les télécharger, quelle version du SDK Android utiliser …

Pour ce faire *Gradle* se base sur un ensemble de fichiers de scripts, écrits dans un langage qui se nomme Groovy.

Lorsque l'on créé un projet avec Android Studio, celui-ci génère plusieurs scripts *Gradle*. Android Studio crée un script *Gradle* pour chaque module du projet, ainsi qu'un script pour le projet entier. Chaque fichier de script présent à un niveau surcharge le fichier équivalent du niveau supérieur.

Configuration au niveau projet :

=> ***build.gradle*** : décrit la configuration de *Gradle* lui-même
=> ***settings.gradle*** : indique les modules que *Gradle* doit traiter
=> ***local.properties*** : indique l'emplacement du SDK sur le poste de travail

Dans le script du projet ***build.gradle***, on trouve les informations partagées par tous les modules du projet.

```
// Top-level build file where you can add configuration options common to all sub-projects/modules.
buildscript {
    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.5.1'

        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        google()
        jcenter()

    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

Le fichier ***build.gradle*** du module contient :

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 28
    defaultConfig {
        applicationId "com.cours.exemple.affichemessage"
        minSdkVersion 23
        targetSdkVersion 28
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
```

```
        }
    }
}

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
}
```

- ✓ La rubrique *apply* indique à *Gradle* quel *plugin* il doit utiliser pour construire l'APK
- ✓ Le bloc *android* regroupe toutes les informations fournies par le développeur pour configurer le projet lors de sa création
- ✓ dans la rubrique *buildTypes*, on pourrait décrire 2 types de *build* : *release* et *debug* si on souhaite les différencier, ce qui n'est pas le cas dans le fichier généré par défaut.
- ✓ dans la dernière rubrique, on trouve les dépendances de l'application

Les ressources d'une application Android

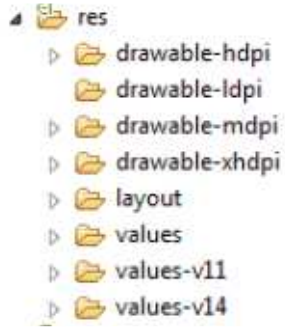
1) Généralités

Les ressources sont l'ensemble des éléments embarqués dans l'application et dont elle a besoin pour fonctionner : les sons, les images, les vidéos ... **Les noms des fichiers de ressource sont obligatoirement écrits en minuscules.** Ils peuvent aussi contenir des chiffres, ou le caractère '_'.

Répertoire `res` Ce répertoire contient les ressources de l'application. Celles-ci sont réparties dans plusieurs sous-répertoires selon leur nature. La liste ci-dessous présentes les principaux. Notons que certains ne sont pas créés par défaut.

`res/drawable/`	Les images sous la forme de fichiers png, jpeg, ou gif ou bien de fichiers XML pour décrire les dessins simples (cercles, carrés ...)
`res/layout/`	Description XML de l'interface graphique
`res/menu/`	Description XML des menus
`res/raw/`	Fichiers généraux : clip audio, fichier CSV ...
`res/values/`	Définitions des constantes : chaînes de caractères, dimensions ...
`res/xml/`	Les autres fichiers XML
`res/minmap`	Les icônes de lancement de l'application déclinées selon différentes densité d'écran

Les applications sont destinées à être utilisées sur un grand nombre de supports différents. Elles doivent s'adapter à ces supports. L'un des moyens de le faire est d'utiliser le concept de ressource. Les ressources sont des fichiers qui sont organisés d'une manière particulière qu'Android est capable de comprendre. Android choisit alors automatiquement le fichier le plus adéquat selon le type de matériel sur lequel s'exécute l'application.



Certains noms de répertoires ont un suffixe (nommé aussi quantificateur, qualificateur ou qualificatif), comme par exemple *hdpi* dans `res/drawable-hdpi`. Ce suffixe indique que ce répertoire de ressources

ne doit être utilisé que dans certaines situations, ici pour les terminaux dotés d'écrans haute résolution (*hdpi = high dot per inch*). Le choix du répertoire à utiliser est effectué automatiquement par le système Android, dès lors que plusieurs répertoires sont présents. Si Android ne trouve pas d'emplacement dont le nom corresponde exactement aux spécifications techniques du terminal, il cherchera parmi les autres répertoires qui existent la solution la plus proche.

D'autres suffixes correspondent à

- ✓ la langue, et éventuellement la région, du système de l'utilisateur. Par exemple : `en`, `fr`, `fr-rFR`, `fr-rCA` ...
- ✓ la taille de l'écran : `small`, `normal`, `large`, `xlarge`
- ✓ l'orientation de l'écran : `port` ou `land`
- ✓ la version d'Android. Il s'agit du niveau de l'API, par exemple : `v11` ou `v14` ...

2) La densité d'écran

Les écrans des terminaux Android ont des caractéristiques très diverses du point de vue de la taille, ou de la résolution. La taille de l'écran peut être comprise entre 3 et 4 pouces pour les plus petits modèles. Les tablettes performantes sont équipées d'écran de 10 pouces ou plus.

Les smartphones peuvent avoir une résolution d'écran de 480 par 800 pixels, alors qu'un terminal haut de gamme aura une résolution pouvant dépasser 1080 par 1920 pixels. On trouve même des résolutions de 2160 par 4096 pixels.

La combinaison de ces 2 informations, taille et résolution d'écran, donne une caractéristique supplémentaire : la **densité de pixel** qui s'exprime en point par pouce (**dot per inch** ou **dpi**). Ainsi un terminal avec un grand écran mais doté d'une faible résolution aura une densité de pixel faible. Au contraire, un smartphone avec un écran de petite taille, et une résolution élevée possèdera une densité écran élevée.

Android classe les écrans en plusieurs catégories :

Catégorie d'écran		Densité moyenne (en dpi)
ldpi	Low dpi	120 pixels par pouce
mdpi	Medium dpi	160
hdpi	High dpi	240
xhdpi	Extra high dpi	320
xxhdpi		480
xxxhdpi		640

Par exemple, si une image a une largeur de 120 pixels, elle s'affichera sur 1 pouce, sur un écran *ldpi*, mais sur seulement 0,5 pouce sur un écran hdpi.

La difficulté pour le programmeur est de gérer correctement l'affichage d'une telle image quel que soit le terminal utilisé. Android facilite la tâche du programmeur grâce au gestionnaire de ressource. Android propose des répertoires spécifiques à chaque catégorie de ressource. Par exemple :

`/res/drawable`	images de dimension standard
`/res/drawable-ldpi`	images utilisées par les écrans de densité 120 dpi (*low*)
`/res/drawable-mdpi`	densité 160 dpi (*medium*)
`/res/drawable-hdpi`	densité 240 dpi (*high*)
`/res/drawable-xhdpi`	densité 320 dpi (*extra-high*)

Le programmeur peut donc, s'il le souhaite, fournir à l'application des images différentes adaptées à chaque densité d'écran. S'il ne personnalise pas les images, le système effectuera une mise à l'échelle des visuels selon la densité de l'écran du terminal.

L'unité density-independant pixel

La multiplicité des densités des écrans introduit un autre problème : la dimension exacte des composants d'une interface. Par exemple, si on prévoit d'afficher un bouton de 200 pixels de long, la longueur effective sera dépendante du terminal qui l'affichera, puisqu'il n'y a pas le même nombre de pixels par pouce pour deux écrans de densité différente.

Pour éviter ce problème, Android fournit une unité de dimension particulière : le **density-independant pixel** (pixel indépendant de la densité). L'unité est notée **dip** (ou **dp**).

Lorsqu'un composant de l'interface a une dimension exprimée en *dpi*, le système effectue une conversion en fonction de la densité effective de l'écran selon la formule :

px = dip * (dpi / 160)

px = dimension en pixel
dip = dimension indiquée par le développeur
dpi = densité de l'écran

Exemple pour un bouton de largeur 100 dip, la dimension en pixel sera la suivante :

Densité écran	Densité	Dimension en pixel
ldpi	120	75
mdpi	160	100
hdpi	240	150
xhdpi	320	200
xxhdpi	480	300
xxxhdpi	640	400

On utilise de préférence l'abréviation **dp** au lieu de **dip** pour éviter les confusions avec l'unité **dpi**.

Unités possibles pour les dimensions

Unité	Description
dp ou **dip**	Unité indépendante de la densité de l'écran (**dp** conseillé)
sp ou **sip**	Unité dépendante de la taille préférée de police de caractères (**sp** conseillé)
px	pixel
in	Inch (1pouce = 2,54 cm)
mm	Millimètre

3) Gérer plusieurs versions d'API

Le gestionnaire de ressources permet également de différencier des ressources selon le niveau de l'API du terminal. Cette possibilité est très utile pour tirer le meilleur parti des styles. En effets, ceux-ci sont très dépendants de la version d'API du système.

Pour caractériser un dossier de ressources par une version d'API, il faut ajouter au nom du dossier le suffixe -v suivi du numéro de l'API. Par exemple, on pourrait définir les répertoires suivants :

```
/res/values-v11/
/res/values-v14/
/res/values-v21/
```

Les valeurs définies dans `values-v11` seront utilisées sur les terminaux avec des API 11, 12 et 13. Les valeurs définies dans `values-v14` seront, quant à elles, utilisées pour les systèmes avec API 14 à 20.

Si on a besoin de connaître le niveau de l'API dans le code Java, on consulte la constante `android.os.Build.VERSION.SDK_INT`

```
if (android.os.Build.VERSION.SDK_INT
                        >= android.os.Build.VERSION_CODES.GINGERBREAD) { ...
```

4) La localisation des applications

Android dispose de facilités pour obtenir rapidement des applications multilingues. Chaque répertoire de ressources peut être caractérisé par le code de la langue à laquelle il est destiné.

Conjointement, il est fortement déconseillé d'utiliser directement les chaînes de caractères, dans les fichiers *layout* ou dans le code Java. Les chaînes de caractères doivent être définies sous la forme de valeurs de type *String*, stockées dans le fichier `strings.xml`.

Pour l'internationalisation, l'idée est de créer un fichier `strings.xml` pour la langue par défaut, et de créer un fichier `strings.xml` par langue supportée. Le fichier par défaut sera placé dans le dossier `res/values`, et les autres dans des répertoires `res/values-[code langue]-[code région]`

Le code langue est un code ISO 639 à 2 lettres.
La région est définie par un code ISO 3166-1-alpha-2 à 2 lettres précédé de la lettre **r**.

Par exemple :	`values-fr`	langue française
	`values-fr-rCA`	langue française au Canada

Exemples de code pour les langues : `en` = anglais, `it` = italien, `es` = espagnol ...

Tous les répertoires de ressources peuvent utiliser la localisation. Par exemple :
`res/drawable-fr-rCA`

Remarque : l'internationalisation ou *internationalization* est souvent abrégée en **i18n** (i, puis 18 caractères, puis n).

5) Généralisation

Android propose d'autres qualificatifs pour les ressources : pour l'orientation de l'écran, ou pour la dimension de l'écran. Le tableau ci-dessous résume les différents qualificatifs :

Propriété	**Exemple de qualificatif ou qualificateur**	**Description**
Langue	`en fr en-rUS`	
Largeur	`sw320dp` `sw480dp`	Plus petite dimension de l'écran `sw` = smallest width (plus petite largeur) sw320dp cible les terminaux ayant une largeur minimale de 320 dp
Orientation de l'écran	`port` `land`	Il est possible de fournir des ressources différentes selon la position de l'écran : portrait ou paysage (*landscape*)
Densité de l'écran	`ldpi mdpi` `hdpi`	
Version d'API	`v7 v11 v14`	

Les différents qualificatifs que l'on peut ajouter aux dossiers de ressources peuvent être combinés. Par exemple :

```
res/drawable-port-hdpi
res/values-fr-v11
res/values-en-v14
```

Il est possible de concaténer plusieurs suffixes. Il y a un ordre à respecter pour les qualificatifs (voir la documentation officielle).

6) Les chaînes de caractères

Généralement, le fichier *strings.xml* débutera par une ligne spécifiant l'encodage souhaité pour les caractères. Par exemple, si l'encodage est `utf-8`, on écrira :

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

1) Gestion des apostrophes

Pour gérer correctement les apostrophes, il faut soit écrire la chaîne entre guillemets, soit utiliser le caractère d'échappement \ devant l'apostrophe.

```
<string name="apostrophe1">"Voici l'exemple d'une apostrophe"</string>
<string name="apostrophe2">Voici l\'exemple d\'une apostrophe</string>
```

2) Ajouter des arguments

On peut ajouter des arguments aux chaînes de caractères afin de les adapter à des valeurs déterminées de manière dynamique.

Par exemple, la chaîne ci-dessous contient 2 arguments. Le premier repéré par **%s** est une chaîne de caractères, le deuxième repéré par **%d** est un entier.

```
<string name="montant_reduction"> %s ! Vous avez droit à une réduction de %d euros </string>
```

Dans le code Java, on peut ensuite remplacer les arguments par des valeurs précises :

```
String message = String.format(
                  getResources().getString(R.string.montant_reduction),
                  "Pierre Dupont", 15);
```

La chaîne *« Pierre Dupont »* ira remplacer l'argument %s, et la valeur 15 remplacera l'argument %d.

Depuis les dernières versions, grâce à une surcharge de la méthode ***getString***, on peut écrire plus simplement :

```
String message = getString(R.string.montant_reduction, "Pierre Dupont", 15);
```

3) Balise HTML

Il est possible d'utiliser des balises HTML pour modifier la présentation des chaînes. Par exemple, les balises **b**, **i, u** sont utilisables.

Par exemple :

```
<string name="message_erreur"> <b>Attention !</b> une erreur s'est produite</string>
```

4) Les tableaux

La balise `string-array` permet de stocker des tableaux de chaînes de caractères.

Par exemple, on pourrait placer dans un fichier *arrays.xml*, le code suivant :

```
<resources>
    <string-array name="couleur">
        <item>Rouge</item>
        <item>Bleu</item>
        <item>Vert</item>
        <item>Noir</item>
        <item>Blanc</item>
        <item>Jaune</item>
    </string-array>
</resources>
```

Dans le code Java, on récupère le tableau de la manière suivante :

```
String[] couleurs = getResources().getStringArray(R.array.couleur);
```

5) Remarque

On peut aussi créer des fichiers ***xml*** pour stocker des entiers et des booléens, sur le même principe que celui des chaînes de caractères. On peut également déclarer des tableaux d'entiers, par exemple.

7) Les dimensions et les couleurs

Il est possible et conseillé de définir les valeurs des marges ou des autres dimensions en tant que ressources.

En effet, il sera parfois nécessaire de laisser des marges entre le bord gauche de l'écran et les *widgets*, ou bien de laisser de l'espace entre les *widgets* dans le sens vertical, ou bien au dessous ou au dessus d'une barre horizontale. Ces valeurs seront identiques pour la totalité des informations affichées. Pour faciliter la mise au point de l'interface, une bonne manière de procéder est de définir ces valeurs en tant que constantes dans un fichier de ressource nommé ***dimens.xml***, par exemple. Notons que le programmeur peut choisir librement le nom du fichier. Ce fichier sera placé dans le dossier ***values***, tout comme le fichier ***strings.xml,*** et il aura la même structure que ce dernier (balise `resources`). A l'intérieur de ce fichier, nous utiliserons la balise `dimen`, comme dans l'exemple ci-dessous :

```
<resources>
    <dimen name="marge_verticale">30dp</dimen>
<resources>
```

Cette valeur sera ensuite employée dans le fichier ***layout.xml*** en suivant le même principe que celui des chaînes de caractères. On écrira donc : `"@dimen/marge_verticale"`.

dp est l'unité recommandée pour la mise en page (taille, marge, espacement ...)
sp (échelle de pixels indépendants) est l'unité recommandée pour la taille des polices

Les couleurs également peuvent être définies dans un fichier de ressource. Il est recommandé d'adopter cette pratique. Par exemple, on pourrait écrire le code suivant dans un fichier de ressources nommé ***colors.xml*** :

```
<resources>
    <color name="red">#F00</color>
<resources>
```

Pour utiliser cette valeur, on écrira donc : `"@color/red"`.

8) Les styles et les thèmes

Notion de style

Il est possible et conseillé de définir des styles pour décrire l'aspect visuel d'un *widget*. L'idée est de regrouper les caractéristiques visuelles d'un *widget* au sein d'un style. Les avantages sont nombreux : le code obtenu sera ainsi plus clair, le style pourra être réutilisé pour plusieurs *widgets*, et on obtiendra plus facilement une application avec une interface homogène.

Un style est une ressource définie dans un fichier XML nommé généralement ***styles.xml*** et placé dans le dossier ***values***.

Exemple d'un fichier de style :

```
<resources>

    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>


    <!-- style de la phrase de titre affichée en haut de chaque onglet -->
    <style name="titre" parent="AppTheme">
        <item name="android:layout_width">match_parent</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:layout_marginBottom">@dimen/marge_verticale</item>
        <item name="android:textColor">@color/couleur_texte_titre</item>
        <item name="android:textSize">@dimen/taille_titre</item>
        <item name="android:textStyle">bold</item>
        <item name="android:gravity">center</item>
        <item name="android:layout_gravity">center</item>
        <item name="android:layout_marginTop">@dimen/marge_verticale</item>
    </style>

    <!-- style des étiquettes invitant à faire une saisie -->
    <style name="etiquette" >
        <item name="android:layout_width">wrap_content</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:textColor">@color/couleur_texte_etiquette</item>
    </style>

    <!-- style des boutons pour valider, effacer , faire un retour .... -->
    <style name="bouton" >
        <item name="android:layout_width">wrap_content</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:background">@color/couleur_bouton</item>
        <item name="android:textSize">@dimen/taille_etiquette</item>
        <item name="android:textColor">@color/couleur_texte_titre</item>
        <item name="android:paddingLeft">@dimen/padding_bouton</item>
        <item name="android:paddingRight">@dimen/padding_bouton</item>
        <item name="android:textAllCaps">false</item>
    </style>


</resources>
```

Tout style possède un nom précisé via l'attribut ***name*** de la balise ***style***. Souvent un style possèdera un thème parent précisé via l'attribut ***parent***. Dans ce cas, le style hérite de toutes les propriétés du thème parent.

Les caractéristiques d'un style sont définies avec des balises ***item***. Chaque ***item*** possède un nom et une valeur. Le nom est un attribut possible pour le *widget* auquel le style sera associé, comme par exemple : **`android:textSize`** ou **`android:textColor.`** La valeur de l'item est soit une valeur littérale comme par exemple *30sp* ou bien un identifiant de ressource comme par exemple **`@dimen/taille_etiquette.`**

Pour appliquer un style à un *widget*, on utilise l'attribut ***style*** :

```
<TextView
      style="@style/etiquette"
      . . . />
```

Notion de thème

Un thème est un ensemble de styles qui s'appliquent à toute une activité ou à toute l'application. Pour appliquer un style à l'ensemble de l'application, il faut utiliser l'attribut ***android:theme*** de la balise ***application*** dans le fichier ***manifest***. Dans le cas où on souhaite appliquer le style seulement à une activité, on procède de même, mais avec l'attribut ***android :theme*** de l'activité.

Exemple de définition d'un style pour tous les boutons d'une application (ou activité) :

On définit d'abord un style pour les boutons comme vu dans la section précédente. Supposons que le style se nomme *bouton*. Ensuite, il faut préciser que ce style doit s'appliquer à tous les boutons. Pour ce faire, on renseigne l'attribut ***buttonStyle*** avec la valeur *bouton*, ceci dans le style que l'on associera à l'application. On obtient donc :

```
<style name="AppTheme" parent="Theme.AppCompat.Light">
      <item name="android:buttonStyle">@style/bouton</item>
</style>
```

L'attribut ***parent*** permet de préciser le style de base sur lequel sont faites des modifications pour obtenir le nouveau style, ici *AppTheme*. Si aucun parent n'est défini, le programmeur devra définir lui-même tous les éléments de style de l'application ou activité, ce qui en général n'est pas souhaitable.

Remarque

Le thème des applications a beaucoup évolué au fil des versions d'Android. Depuis la version 4, c'est le thème ***MaterielDesign*** qui s'impose. Le thème ***AppCompat*** est celui qu'il faut utiliser pour les versions antérieures à la 4 si l'on souhaite obtenir un thème ressemblant à ***MaterialDesign***.

Pour plus d'informations sur la notion de ressources,
consulter le site **Android Developer**, dans la partie **documentation**, dans l'onglet **Guide**, à la rubrique ***App Resources***.

Les interfaces utilisateur - Les bases

I) Quelques principes de base sur les layouts et widgets

Comme indiqué précédemment, on préfère définir un fichier de positionnement (*layout*) codé en XML, plutôt que de créer et de positionner les composants (*widgets*) dans le fichier contenant le code Java. Bien sûr ceci n'est réalisable que pour la partie statique de l'interface. Les principaux avantages de cette organisation sont :

- ✓ le code est plus lisible en XML qu'en Java,
- ✓ séparation entre la partie vue et les traitements, le code obtenu est plus visible (il s'agit d'une partie de la mise en œuvre du modèle MVC)
- ✓ on peut éventuellement utiliser les utilitaires pour concevoir l'interface graphique, ou au moins pour la visualiser lorsqu'elle est en cours d'élaboration
- ✓ XML a l'avantage d'être bien connu de la plupart des développeurs

Chaque fichier XML contient une arborescence d'éléments précisant le *layout* à utiliser et les *widgets* qu'il contient.

Rappel : Les fichiers XML décrivant le positionnement des *widgets* sont situés dans le dossier `res/layout`.

Exemple : description d'un bouton

Par exemple, pour décrire un *widget* bouton, on écrira le code HTML suivant :

```
<Button
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="@15dip"
    android:text="@string/boutonValider"
    android:background="#000000"
    android:textColor="#FFFAFA"
    android:onClick="clicSurValider"    />
```

Le nom de l'élément XML est ***Button***, c'est aussi le nom de la classe Java correspondant à ce composant. Cette règle est valable pour tous les composants.

Les autres attributs donnent les propriétés du bouton :

`android:text`	précise le texte à afficher sur le bouton (mieux vaut utiliser une constante définie dans le fichier de ressource `strings.xml`)
`android:layout_width`	précise la largeur du bouton (ici celle du conteneur parent)
`android:layout_heigth`	précise la hauteur du bouton (celle du texte contenu dans le bouton)
`android:layout_marginTop`	marge supérieure à laisser au dessus du bouton.
`android:background`	couleur de fond du bouton
`android:textColor`	couleur du libellé du bouton
`android:onClick`	méthode appelée automatiquement lors d'un clic sur le bouton. Cette méthode doit être définie dans la classe Java associée au fichier XML (une classe *activité*)

Lien entre le *widget* défini dans le fichier *layout* et le code Java de l'activité

Si le *widget* défini dans le fichier XML a besoin d'être référencé dans le code Java (par exemple ce serait le cas pour une zone de saisie, car on souhaitera accéder au texte entré par l'utilisateur), il doit obligatoirement posséder un identifiant. Pour ce faire, il faut préciser une valeur pour l'attribut **`android:id`**, afin de donner un nom (identifiant) au *widget*. On doit alors respecter la syntaxe suivante :

```
android:id="@+id/un_bouton"
```

Le fichier XML est utilisé, par l'environnement de développement, pour définir automatiquement des éléments de la classe R. En particulier, un identifiant numérique sera créé dans cette classe, son nom sera `R.id.un_bouton`. C'est via cet identifiant que le *widget* pourra être référencé dans le code Java.

Dans le code Java, pour accéder au *widget*, on utilise la méthode `findViewById(R.id.un_bouton)` qui renvoie un résultat de type *View* (voir plus loin). Il faut convertir cet objet en une instance de *Button* en écrivant :

```
(Button) findViewById(R.id.un_bouton)
```

Remarque : création et initialisation d'un *widget* dans le code Java

Il est possible de créer et de fixer les propriétés d'un *widget* dans le code Java de l'activité. Par exemple:

```
public class ActivitePrincipale extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        TextView etiquette = new TextView(this);
        etiquette.setText("Nouveau message bonjour !");
        setContentView(etiquette);
    }
}
```

Il est préférable d'utiliser cette possibilité seulement pour la partie dynamique de l'interface.

Largeur et hauteur d'un *widget*

Pour spécifier la largeur et la hauteur d'un *widget*, on peut utiliser les valeurs prédéfinies :

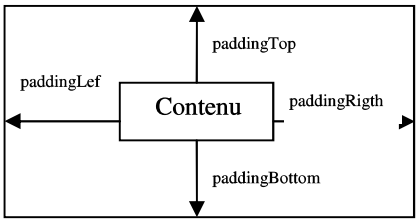
- `fill_parent` ou **`match_parent`** (`match_parent` est la nouvelle valeur depuis Android 2.2) l'élément graphique remplit tout l'espace de son parent (son contenant)
- **`wrap_content`** l'élément n'utilisera que l'espace nécessaire pour s'afficher dans son parent
- ou bien une valeur précise accompagnée de son unité.

Les 2 unités qu'il est préférable d'utiliser sont (voir aussi le chapitre sur *les ressources*) :

- **dp** ou **dip** unité indépendante de la résolution d'écran (*dot independant pixel*)
- **sp** cette unité est également indépendante de la résolution d'écran. Elle est utilisée pour définir une taille de police.

Notions de *padding* et de marge d'un *widget*

On peut aussi définir une marge interne pour chaque *widget*, autrement dit l'espacement entre la bordure du *widget* et son contenu : le *padding*.



En fait, il y a 4 valeurs de marges possibles :
`paddingTop, paddingBottom, paddingLeft, paddingRight`.

Si le *padding* est identique dans les 4 directions, on utilise tout simplement l'attribut `padding` :

```
android:padding="10 dip"
```

Si les valeurs des *paddings* varient selon les directions, on écrira :

```
android:paddingTop="10 dip"
android:paddingBottom="15 dip"
android:paddingLeft="20 dip"
android:paddingRight="30 dip"
```

Il ne faut pas confondre le *padding* (marge interne) et la marge : la marge correspond à l'espace à laisser autour du *widget*. Les remarques pour les unités sont valables aussi pour la marge.

II) Utilisation des widgets de base

Hiérarchie d'héritage entre les composants (*widgets* et *layouts*)

```
View
      ViewGroup (*)
            FrameLayout
                  DatePicker
                  TimePicker
                  ScrollView
            LinearLayout
                  TableLayout
                  TableRow
                  RadioGroup
            RelativeLayout
            GridLayout
            AdapterView (*)
                  AbsListView (*)
                        ListView
                        GridView
      TextView
            EditText
            Button
                  CompoundButton (*)
                        CheckBox
                        ToggleButton
                        RadioButton
                        Switch
      ImageView
            ImageButton
```

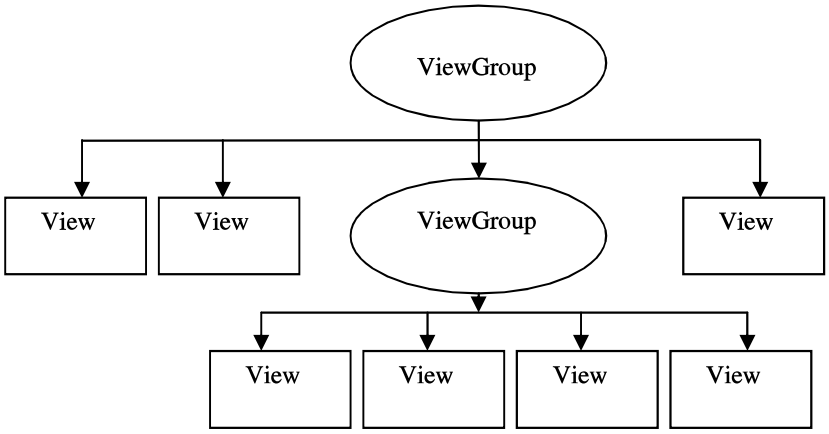
Légende :	**en gras**	les *widgets* (éléments d'interaction)
	encadré	les conteneurs ou groupes
	(*)	classe abstraite

Remarques

- *View* est la classe de base pour tous les éléments visuels des interfaces
- Les groupes de vues ou *ViewGroup* sont des extensions de vues (*ViewGroup* hérite de *View*) qui ont la particularité de pouvoir contenir plusieurs vues filles
- Un *ViewGroup* peut contenir des *View*, et par conséquent des *ViewGroup*
- Les *layouts* (ou gestionnaires de positionnement) héritent de *ViewGroup*, et *ViewGroup* hérite de *View*.
- Un *layout* pourra donc contenir d'autres *layouts*

Organisation d'une interface

A la base de l'interface, on trouvera toujours un *ViewGroup*. Ce *ViewGroup* sera un gestionnaire de positionnement (ou *layout*). Dans le *ViewGroup* racine, on imbriquera des *View* ou d'autres *ViewGroup* qui eux-mêmes pourront contenir des *View* ou d'autres *ViewGroup*.



Quelques attributs de *View*

	Attribut	Rôle
View	`android:padding`	Espacement entre la bordure du *widget* et son contenu
	`android:paddingTop`	Espacement avec le bord supérieur
	`android:paddingBottom`	Espacement avec le bord inférieur
	`android:paddingLeft`	Espacement avec le bord gauche
	`android:paddingRight`	Espacement avec le bord droit
	`android:background`	Couleur de fond (une valeur en hexadécimal)
	`android:id`	Spécifie un identifiant pour le composant
	`android:onClick`	Nom de la méthode Java invoquée lors d'un clic dans ce composant
	`android:theme`	Spécifie un thème associé à la vue
	`android:focusable`	Spécifie si le composant peut être activé (avoir le focus)

Quelques attributs de *ViewGroup*

Ces attributs peuvent être utilisés dans les vues enfants du *ViewGroup* qui les contient. Mais c'est le *ViewGroup* qui gère la prise en compte des marges spécifiées.

	Attribut	Rôle
ViewGroup	`android:layout_marginTop`	Marge au dessus du composant
	`android:layout_marginBottom`	Marge au dessous du composant
	`android:layout_marginLeft`	Marge à gauche du composant
	`android:layout_marginRight`	Marge à droite du composant

Dénomination de quelques *widgets*

Les *widgets* sont les éléments d'interaction de l'interface utilisateur. Les principaux sont présentés dans les sections suivantes de ce chapitre.

	Rôle	Nom du *widget*
	Texte du titre Image Boutons radio Texte et zone de saisie Case à cocher Boutons	***TextView*** ***ImageView*** ***RadioButton*** ***TextView*** ***EditText*** ***CheckBox*** ***Button***

A) Les labels ou étiquettes - `TextView`

Les `TextView` sont utilisés pour afficher du texte à l'intention de l'utilisateur, et en option lui permettre de le modifier. Cependant, si l'utilisateur est amené à modifier le texte, il est préférable d'utiliser le *widget* `EditText` (voir plus loin). Cette possibilité de modifier un `TextView` explique la présence de plusieurs attributs pour décrire les possibilités de modifications.

	Attribut	Rôle
TextView	`android:typeface`	Police du label
	`android:textStyle`	Gras, italique (`bold, italic, bold_italic`)
	`android:textColor`	Couleur au format RGB hexadecimal (#FF0000 pour rouge, par exemple). Couleur du texte
	`android:textSize`	Taille du texte. La dimension recommandée est `sp` (scaled-pixel). Exemple : `"15sp"`
	`android:gravity`	Spécifie comment aligner le texte contenu dans le TextView. Une valeur (ou plusieurs séparées par |) parmi celles-ci : `top, bottom, left, right, center_vertical, center_horizontal, center, fill_vertical, fill_horizontal, fill ...`

	Attribut	Rôle
TextView (pour modifier)	`android:autoText`	Pour indiquer si le champ doit fournir une correction automatique d'orthographe. La valeur peut donc être `"true"` ou `"false"`.
	`android:capitalize`	Pour demander au champ de mettre automatiquement la première lettre en majuscule. La valeur peut donc être `"true"` ou `"false"`.
	`android:digits`	le champ n'acceptera que certains caractères spécifiés dans une chaîne de caractères. Ces caractères ne sont pas obligatoirement des chiffres.
	`android:password`	Le champ n'affichera que des points à la place des caractères saisis. La valeur peut donc être `"true"` ou `"false"`.
	`android:singleLine`	Pour indiquer si la saisie peut se faire sur une ou plusieurs lignes (booléen). La valeur peut donc être `"true"` ou `"false"`.

	`android:inputType`	Pour autoriser seulement certains types de saisie.
		La valeur est une chaîne de caractères contenant une classe et éventuellement des modificateurs, séparés par le caractère | (tube). Schéma :
		`android:inputType = "`*`une_classe`* `| modificateur1 |`
		`modificateur2` `|` *`modificateur3`*`"`
		
		Les classes possibles sont :
		`text, number, phone, datetime, date, time`
		
		Exemples de modificateurs :
		`numberDecimal numberSigned`
		`textEmailAddress textUri textMultiLine ...`
		
		Exemple :
		`android:inputType = "text|textEmailAddress"`
		
		Remarque : la valeur `"none"` signifie que le texte de la zone n'est pas éditable (modifiable)

	`android:hint`	Pour afficher un texte d'indication
	`android:lines`	Nombre exact de lignes à afficher
	`android:maxLines`	Nombre maximum de lignes
	`android:maxLength`	Nombre maximum de caractères (longueur maximum)

Dans le code Java, on pourra modifier le texte d'un *widget* ***TextView*** (et donc aussi celui d'un ***EditText***), en lui appliquant la méthode ***setText***. Celle-ci possède plusieurs surcharges. Les 2 principales sont :

setText(int resId) on donne en paramètre l'identifiant d'une ressource de type chaîne de caractères définie généralement dans le fichier *strings.xml*

setText(CharSequence texte) on donne en paramètre la chaîne de caractères

Note : *CharSequence* est une interface implémentée par la classe *String*.

B) Les boutons - `Button`

`Button` est une sous-classe de `TextView`. Depuis la version 1.6, il y a 2 approches pour associer un écouteur à un bouton :

- ✓ méthode classique (détaillée plus loin) avec une interface écouteur de bouton
- ✓ méthode plus récente qui consiste à utiliser l'attribut `android:onClick` dans l'élément `Button` du fichier XML

	Attribut	Rôle
Button	`android:onClick`	Pour indiquer le nom de la méthode qui sera appelée automatiquement si on clique sur le bouton. Cette méthode doit être définie dans la classe correspondant à l'activité associée à la vue contenant le bouton.

C) Les images - `ImageView` et `ImageButton`

Il s'agit des équivalents en image de `TextView` et de `Button`. `ImageButton` est une sous-classe de `ImageView`.

		Attribut	Rôle
ImageButton	**ImageView**	`android:src`	Image à associer au widget. Il s'agit généralement d'une ressource graphique, donc on aura par exemple : `android:src= "@drawable/une_image"`
		`android:adjustViewBounds`	Un booléen égal à vrai si la taille de l'image doit être ajustée à la surface disponible.

D) Les champs de saisie - `EditText`

`EditText` est une sous-classe de `TextView`.

Méthodes Java

- La méthode ***getText()*** renvoie le texte présent dans le *widget* (donc celui tapé par l'utilisateur)
- la méthode ***setText*** avec en argument soit la chaîne à écrire soit une référence à un identifiant permet de modifier le texte affiché. Cette méthode est héritée de la classe ***TextView***

E) Les cases à cocher - `CheckBox`

`CheckBox` est une sous-classe de `Button`. Dans le code Java, on pourra utiliser les méthodes suivantes :

`boolean isChecked()`	pour savoir si la case est cochée ou décochée
`void setChecked(boolean)`	pour forcer la case dans l'état coché (ou décochée)
`void toggle()`	pour inverser l'état de la case

La classe Java qui correspond à l'activité peut implémenter l'interface `CompoundButton.OnCheckedChangeListener` de manière à écouter les changements d'état de la case à cocher. C'est la méthode `onCheckedChanged` qui est appelée automatiquement lors d'un tel changement.

CompoundButton (classe abstraite)	Attribut	Rôle
	`android:checked`	Indique l'état initial de la case, cochée ou pas. Exemple : `android:checked = "true"`
	`android:button`	Dessin utilisé pour le graphisme du bouton
	`android:tint`	Teinte appliquée au graphisme du bouton

F) Les switch - `Switch`

Ils existent depuis Android 4.0. C'est une variante d'une case à cocher : un *switch* a 2 états que l'utilisateur peut faire glisser avec son doigt, comme un interrupteur. Il peut également changer son état en le touchant, comme une case à cocher.

Dans le code Java, on pourra utiliser les méthodes suivantes :

`CharSequence getTextOn()`	renvoie le texte associé à l'état "oui" du *switch*
`CharSequence getTextOff()`	renvoie le texte associé à l'état "non" du *switch*
`void setChecked()`	pour forcer le *switch* à l'état "oui"

Tout comme pour les cases à cocher, l'interface `CompoundButton.OnCheckedChangeListener` et la méthode `onCheckedChanged` peuvent être utilisées pour gérer les changement d'état au sein du code Java.

G) Les boutons radio - `RadioButton`

Les boutons radio d'Android ont 2 états tout comme les *switch* et les cases à cocher. Une particularité spécifique est qu'ils peuvent être regroupés au sein d'un groupe de manière à ce qu'un seul d'entre eux puisse être dans l'état sélectionné, à un instant donné. Tout ce que l'on peut faire avec une case à cocher est réalisable avec un bouton radio.

La plupart du temps les *widgets* `RadioButton` seront placés dan un conteneur `RadioGroup` qui permet de lier les boutons afin qu'un seul soit sélectionné. Un `RadioGroup` est en fait un *layout*, mais il n'est utilisé que pour disposer des boutons radio. Le code XML aura la structure suivante :

```
<RadioGroup
   on écrira ici les propriétés du groupe de boutons radio
   . . .

   <RadioButton
         on écrira ici les propriétés du premier bouton radio
         . . .                  />
   <RadioButton
         on écrira ici les propriétés du deuxième bouton radio
         . . .                   />
   <RadioButton
         on écrira ici les propriétés du troisième bouton radio
         . . .                  />
</RadioGroup>
```

Exemple : afficher 2 boutons radio pour choisir "Madame" ou "Monsieur". Les 2 boutons seront placés horizontalement, côte à côte.

```
<RadioGroup
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">
    <RadioButton
        android:id="@+id/madame"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:checked="true"
        android:text="@string/chaine_madame"
        android:onClick="onRadioButtonClicked"/>
    <RadioButton
        android:id="@+id/monsieur"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/chaine_monsieur"
        android:onClick="onRadioButtonClicked"/>
</RadioGroup>
```

Si l'on souhaite qu'une méthode soit appelée automatiquement lorsque le bouton radio est sélectionné, il faut prévoir, comme indiqué ci-dessus, la propriété `android:onClick`.

Lien avec le code Java

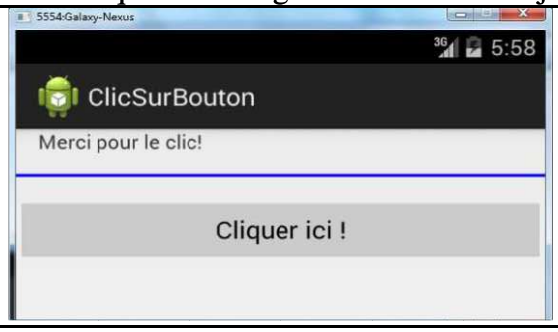
En affectant un identifiant `android:id` au `RadioGroup` dans le fichier XML, ce groupe devient accessible au code Java. On peut alors appliquer les méthodes suivantes :

`void check(int id)`	pour faire en sorte qu'un bouton soit coché, connaissant son identifiant
`void clearCheck()`	pour décocher tous les boutons du groupe
`int getCheckedRadioButtonId()`	pour obtenir l'identifiant du bouton radio actuellement coché (ou -1 si aucun bouton radio n'est coché)

III) Intercepter un clic sur un bouton

Nous disposons de 2 techniques pour intercepter le clic sur un bouton : l'une qui est analogue à ce que nous pourrions faire dans le langage Java standard, l'autre qui est spécifique à Android. La deuxième technique est légèrement plus simple à programmer. Par contre elle a l'inconvénient d'augmenter la dépendance entre la définition du fichier décrivant la vue (code XML) et celui définissant l'activité (code Java).

Nous illustrerons ces possibilités sur l'exemple suivant :

Au lancement de l'application, on doit voir apparaître :	Quand l'utilisateur aura cliqué sur le bouton, on souhaite que le message affiché soit mis à jour :
	

On définit les valeurs des chaînes de caractères dans le fichier ***strings.xml***

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

    <string name="app_name">ClicSurBouton</string>
    <string name="message1">Essai clic sur bouton !</string>
    <string name="premier_bouton"> Cliquer ici !</string>
    <string name="apres_clic">Merci pour le clic!</string>

</resources>
```

Remarque : La gestion des événements qui se produisent sur les autres *widgets* fait appel aux mêmes notions que la gestion d'un clic sur un bouton.

A) Gestion de l'événement entièrement dans le code Java.

Dans cette version, on va associer au bouton un écouteur pour l'événement *clic sur le bouton*. Cet écouteur doit être une classe qui implémente l'interface ***View.OnClickListener.*** Cette dernière contient une méthode nommée ***onClick(View uneVue)***. Cette méthode doit obligatoirement être implémentée puisque c'est elle qui est appelée automatiquement lors du clic sur le bouton.

Nous envisageons 2 possibilités : l'activité est son propre écouteur d'événements sur le bouton ou bien l'écouteur est défini via une classe anonyme.

On définit la vue dans le fichier ***layout :***

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="${relativePackage}.${activityClass}"
    android:orientation="vertical">

    <TextView
        android:id="@+id/message"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="15dip"
        android:text="@string/message1" />

     <View
        android:layout_width="match_parent"
        android:layout_height="2dip"
        android:layout_marginTop="15dip"
        android:background="#0000FF" />

     <Button
        android:id="@+id/le_bouton"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="15dip"
        android:text="@string/premier_bouton" />

</LinearLayout>
```

Exemple 1 : la classe est son propre écouteur d'événement

La classe qui hérite de ***Activity*** et qui est associée à la vue contenant le bouton doit implémenter l'interface ***View.OnClickListener***. Cette interface contenant une méthode nommée ***onClick(View uneVue)***, la classe activité doit bien sûr l'implémenter, obligatoirement.

La classe Java qui décrit l'activité est la suivante :

```
/*
 * Exemple : comment gérer un clic sur un bouton ?
 */
import  . . .

/**
 * Cette activité illustre la gestion d'un clic sur un bouton.
 * La technique utilisée ici est la suivante : on associe un écouteur
 * au bouton dans le code Java.
 * La classe implémente donc l'interface OnClickListener et le traitement
 * à réaliser lors du clic est défini dans la méthode onClick
 * @author LP MMS
 *
 */
public class ActivitePrincipale extends Activity implements OnClickListener  {

    /** Texte affiché sur la vue */
    private TextView etiquette;
```

```
    /** Bouton présent sur la vue */
    private Button bouton;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // on associe une vue (fichier activite_principale.xml)
        setContentView(R.layout.activite_principale);

        // on accède aux widgets définis dans la vue
        etiquette = (TextView) findViewById(R.id.message);
        bouton = (Button) findViewById(R.id.le_bouton);

        // on associe un écouteur au bouton
        bouton.setOnClickListener(this);
    }

    /**
     * Méthode exécutée automatiquement lors du clic sur le bouton
     */
    public void onClick(View view) {

        // on modifie le texte affiché sur l'étiquette
        etiquette.setText(R.string.apres_clic);
    }
}
```

Exemple 2 : l'écouteur est défini via une classe anonyme

Dans cette version, la méthode ***onClick*** n'est plus définie en tant que membre de la classe activité, mais est définie en tant que méthode d'une classe anonyme jouant le rôle d'écouteur de l'événement *clic sur le bouton*. La classe anonyme est donnée directement en tant qu'argument de la méthode ***setOnClickListener***.

Dans la classe *ActivitePrincipale*, il faut donc supprimer la méthode ***onClick*** et remplacer la ligne :

```
// on associe un écouteur au bouton
bouton.setOnClickListener(this);
```

par le code suivant :

```
/*
 * on associe un écouteur au bouton sous la forme d'une
 * classe anonyme qui implémente l'interface OnClickListener
 * et qui définit la méthode onClick
 */
bouton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

        // on modifie le texte affiché sur l'étiquette
        etiquette.setText(R.string.apres_clic);
    }
});
```

B) **Dans l'élément *Button* du fichier XML décrivant la vue, on inclut un attribut `android:onClick`**

Cette manière de procéder est plus simple que la précédente et a été introduite depuis la version 1.6 d'Android. Précisément, nous procèderons ainsi :

✓ dans l'élément ***Button*** du fichier layout XML, on inclut un attribut `android:onClick` qui spécifie le nom de la méthode qui sera invoquée automatiquement lors du clic sur le bouton :

```
<Button
    android:id="@+id/le_bouton"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="15dip"
    android:text="@string/premier_bouton"
    android:onClick="clicSurBouton"/>
```

✓ dans la classe Java qui correspond à l'activité, on écrit une méthode publique qui a en paramètre un objet de type ***View*** et qui renvoie un résultat ***void***. On nomme cette méthode comme on l'a indiqué dans la propriété ***onClick*** du bouton :

```
/*
 * Exemple : comment gérer un clic sur un bouton ?  avec une propriété du bouton
 */
package com.example.clicsurbouton2;
import . . .

/**
 * Cette activité illustre la gestion d'un clic sur un bouton.
 * La technique utilisée ici est la suivante : dans une propriété du
 * widget Button (dans le fichier décrivant la vue), on a indiqué le
 * nom de la méthode qui sera invoquée automatiquement lors du clic
 * sur le bouton : clicSurBouton
 * @author LP MMS
 */
public class MainActivity extends Activity {

    /** Texte affiché sur la vue */
    private TextView etiquette;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // on accède aux widgets définis dans la vue
        etiquette = (TextView) findViewById(R.id.message);
    }

    /**
     * Méthode exécutée automatiquement lors du clic sur le bouton
     */
    public void clicSurBouton(View view) {

        // on modifie le texte affiché sur l'étiquette
        etiquette.setText(R.string.apres_clic);
    }
}
```

Cette deuxième technique pour gérer les événements *clic sur bouton* simplifie d'autant plus le code à écrire que l'interface comporte un grand nombre de boutons.

IV) Les conteneurs (ou gestionnaires de mise en forme)

Les conteneurs (ou ***layout***) permettent de gérer la disposition ou autrement dit le positionnement d'un ensemble de *widgets* et éventuellement des conteneurs fils. En conséquence, une particularité importante à retenir est qu'un *layout* peut contenir d'autres *layouts*. On peut les imbriquer.

A) FrameLayout

C'est le plus simple des gestionnaires. Il empile les vues filles les unes sur les autres. On l'utilisera par exemple pour afficher des onglets (voir un prochain chapitre).

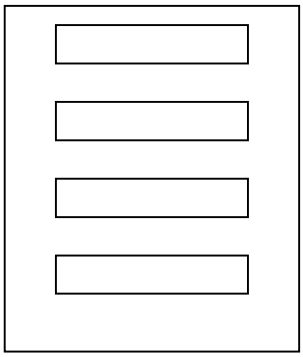
B) LinearLayout

C'est un modèle reposant sur des boîtes. Les boîtes correspondent aux *widgets* ou aux conteneurs fils. Ceux-ci peuvent être alignés en colonnes ou en lignes.

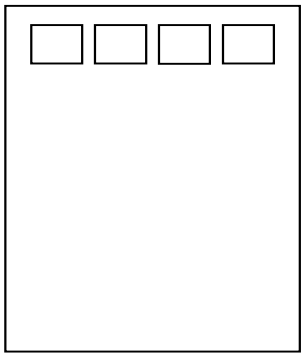
Pour configurer un **LinearLayout**, on peut agir sur 5 paramètres :

- ✓ orientation,
- ✓ modèle de remplissage,
- ✓ poids,
- ✓ gravité,
- ✓ marge.

L'orientation se définit au niveau du **LinearLayout** lui-même avec la propriété suivante :
`android:orientation` La valeur peut être `horizontal` ou `vertical`



Alignement de 4 *widgets* en **vertical**



en **horizontal**

Dans le code Java, au cours de l'exécution de l'activité, on peut aussi fixer l'orientation du ***LinearLayout***:
`setOrientation()` on donne en paramètre la constante HORIZONTAL ou VERTICAL

Les 4 autres paramètres sont spécifiés en tant que propriétés pour chacun des *widgets* contenus dans le **LinearLayout**.

Modèle de remplissage `ViewGroup.LayoutParams`	
`android:layout_width`	On peut préciser pour la largeur : Une dimension précise (pas la meilleure solution) `wrap_content` le *widget* doit occuper sa place naturelle (si trop grand, Android coupera) `fill_parent` le *widget* occupera tout l'espace disponible de son conteneur (notons que la constante `fill_parent` a été remplacée par `match_parent`, dans les versions récentes d'Android)
`android:layout_height`	Idem pour la hauteur
Poids `LinearLayout.LayoutParams`	
`android:layout_weight`	Proportion d'espace libre attribué au *widget*. Par exemple, si 2 *widgets* sont placés, et si le premier a un poids de 1, et le deuxième un poids de 2, le deuxième occupera 2 fois plus d'espace que le premier.
Gravité `LinearLayout.LayoutParams`	
`android:layout_gravity`	Par défaut, les widgets s'alignent à partir de la gauche et en haut. Pour modifier ce comportement, on peut utiliser les valeurs : `left` `right` `center_horizontal` `center_vertical`
Marges `ViewGroup.MarginLayoutParams`	Par défaut, les *widgets* sont serrés les uns contre les autres. Pour ajouter de l'espace entre eux, on utilise la notion de marge.
`android:layout_margin`	Pour spécifier une marge identique sur les 4 côtés
`android:layout_marginTop`	Pour spécifier une marge au dessus du *widget*
`android:layout_marginBottom`	Marge en dessous
`android:layout_marginLeft`	Marge à gauche
`android:layout_marginRight`	Marge à droite

Remarque à propos des poids

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

- ✓ initialiser à 0 les propriétés `android:layout_width` de tous les *widgets* du *layout*
- ✓ initialiser avec des pourcentages adéquats les propriétés `android:layout_weight` de tous les *widgets* du *layout*
- ✓ vérifier que la somme des poids est égale à 100 (ou toute autre valeur)

Remarque à propos de `RadioGroup`

`RadioGroup` hérite de ***LinearLayout***. Donc les propriétés telles que `android:orientation` sont valables aussi pour un `RadioGroupe`.

C) RelativeLayout

Ce gestionnaire place les *widgets* relativement les uns par rapport aux autres *widgets* et par rapport au conteneur parent. On peut préciser que le *widget* X doit être placé en dessous et à gauche du *widget* Y, ou bien préciser que le bord inférieur du *widget* Z doit être aligné avec le bord inférieur du conteneur ...

Positions relatives à un conteneur (les plus simples à décrire)

Elles sont précisées grâce aux propriétés suivantes qui prennent la valeur `true` ou `false`. Ces propriétés sont définies dans la classe *RelativeLayout.LayoutParams* Il faut les appliquer aux *widgets*.

Attribut	Rôle (la valeur est `true` ou `false`)
`android:layout_alignParentTop`	Le haut du *widget* doit être aligné avec celui du conteneur
`android:layout_alignParentBottom`	Le bas du *widget* doit être aligné avec celui du conteneur
`android:layout_alignParentLeft`	Le bord gauche du *widget* doit être aligné avec le bord gauche du conteneur
`android:layout_alignParentRight`	Le bord droit du *widget* doit être aligné avec le bord droit du conteneur
`android:layout_centerHorizontal`	Le *widget* doit être centré horizontalement dans son conteneur
`android:layout_centerVertical`	Le *widget* doit être centré verticalement dans son conteneur
`android:layout_centerInParent`	Le *widget* doit être centré horizontalement et verticalement dans son conteneur

Positions relatives aux autres widgets

Supposons que l'on ait associé à un *widget* **A** une propriété `android:id="@+id/widget_a"` pour lui attribuer un identifiant.

Pour faire référence à celui-ci, dans un autre *widget*, on écrira `"@id/widget_a"`.

Attribut	Rôle
`android:layout_above`	Le *widget* doit être placé au dessus de celui référencé par la propriété
`android:layout_below`	Le *widget* doit être placé au dessous de celui référencé par la propriété
`android:layout_toLeftOf`	Le *widget* doit être placé à gauche de celui référencé par la propriété
`android:layout_toRightOf`	Le *widget* doit être placé à droite de celui référencé par la propriété
`android:layout_alignTop`	Le haut du *widget* doit être aligné avec le haut du *widget* référencé par la propriété
`android:layout_alignBottom`	Le bas du *widget* doit être aligné avec le bas du *widget* référencé par la propriété
`android:layout_alignLeft`	Le bord gauche du *widget* doit être aligné avec le bord gauche du widget référencé par la propriété
`android:layout_alignRight`	Le bord droit du widget doit être aligné avec le bord droit du widget référencé par la propriété
`android:layout_alignBaseline`	Les lignes de base des 2 widgets doivent être alignées Utile pour aligner un label et une zone de saisie.

Ordre d'évaluation

Depuis version 1.6, Android effectue 2 lectures du fichier XML. Il est donc possible de faire référence à un *widget* alors qu'il n'a pas encore été défini.

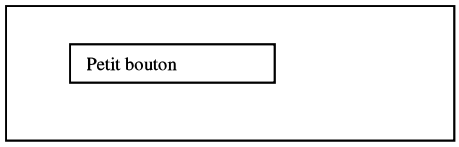
Exemple (Nous aurions pu utiliser un gestionnaire plus simple. Mais le but est d'illustrer l'usage de *RelativeLayout*)

Tous les *widgets* ont une hauteur fixée à **`wrap_content`**, et une largeur également. Sauf le champ de saisie de l'URL qui a pour valeur **`match_parent`**.

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" >

    <TextView
        android:id="@+id/labelURL"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/demande_URL"
        android:layout_alignBaseline="@+id/saisie"
        android:layout_alignParentLeft="true"/>

    <EditText
        android:id="@id/saisie"
        android:hint="@string/indication_URL"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_toRightOf="@id/labelURL"
        android:layout_alignParentTop="true"/>

    <Button
        android:id="@+id/ok"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/bouton_OK"
        android:layout_below="@id/saisie"
        android:layout_alignRight="@id/saisie"/>

    <Button
        android:id="@+id/annuler"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/bouton_annuler"
        android:layout_toLeftOf="@id/ok"
        android:layout_alignTop="@id/ok"/>
</RelativeLayout>
```

Recouvrement de *widgets*

`RelativeLayout` dispose d'une possibilité supplémentaire : un *widget* peut recouvrir un autre. Par exemple, on pourrait placer un petit bouton, dans un grand, ou bien placer un `EditText` dans une image.



Le grand bouton sera configuré pour obtenir tout l'espace disponible, alors que les propriétés du petit indiqueront que la taille doit être adaptée au contenu. Les 2 boutons pourront réagir à des clics.

D) TableLayout

Il permet de positionner les *widgets* sur une grille. On peut définir le nombre de lignes et de colonnes. Celles-ci peuvent s'agrandir ou se réduire en fonction de leur contenu.

`TableLayout` fonctionne conjointement au conteneur `TableRow` qui décrit les rangées (les lignes). `TableLayout` contrôle le comportement global du conteneur. Les widgets eux-mêmes sont placés dans un ou plusieurs `TableRow`, à raison d'un par ligne de la grille.

Placement des cellules dans les lignes

C'est Android qui détermine automatiquement le nombre de colonnes du ***TableLayout***, en fonction du nombre de *widgets* de la ligne qui en contient le plus, ou des indications données par le programmeur. Par exemple, si le ***TableLayout*** contient 3 lignes, l'une avec 2 *widgets*, l'autre avec 3, et une autre avec 4, Android déduira qu'il y a 4 colonnes (sauf mention contraire du programmeur).

Cependant, un *widget* peut occuper plusieurs colonnes. Cette information est donnée avec la propriété **`android:layout_span`**, et en spécifiant le nombre de colonnes sur lesquelles doit s'étendre le *widget* (ressemblance avec *colspan* des tableaux HTML).

On peut aussi décider de placer un *widget* sur une colonne précise avec la propriété **`android:layout_column.`** Si cette propriété n'est pas mentionnée, le *widget* ira sur la première colonne disponible. Les colonnes sont numérotées à partir de 0.

Fils de TableLayout qui ne sont pas des lignes

On peut placer des *widgets* entre les lignes. ***TableLayout*** se comportera alors un peu comme un conteneur ***LinearLayout*** avec une orientation verticale. Les largeurs de ces *widgets* seront automatiquement fixées à `match_parent` pour remplir le même espace que la ligne la plus longue.

Exemple d'utilisation : pour tracer une ligne horizontale entre 2 lignes formées de *widgets*, on utilisera un *widget* `View`.

Réduire, étirer et refermer

Par défaut, la largeur de chaque colonne sera celle du *widget* le plus large qu'elle contient. On peut modifier ce comportement.

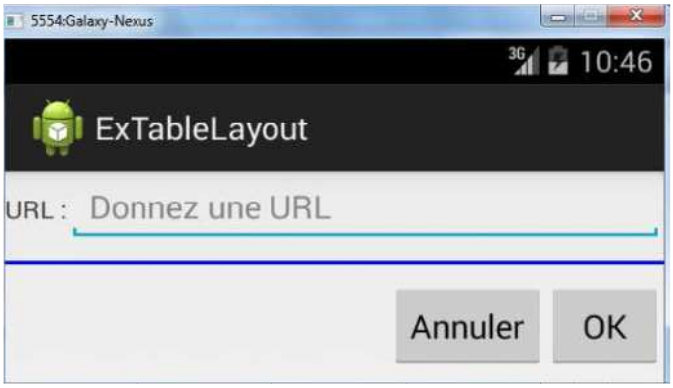
La propriété **`android:stretchColumns`** de l'élément ***TableLayout*** peut avoir pour valeur un seul numéro de colonne, ou une liste de numéros séparés par des virgules. Ces colonnes seront étirées pour occuper tout l'espace disponible de la ligne.

Inversement, la propriété `android:shrinkColumns` de ***TableLayout*** permet de réduire la largeur effective des colonnes en découpant leur contenu sur plusieurs lignes (pour éviter qu'un contenu trop long pousse certaines colonnes à droite de l'écran).

On peut aussi refermer (c'est-à-dire cacher) des colonnes avec `android:collapseColumns` (comme précédemment on indique un numéro de lignes ou plusieurs).

A partir du code Java, de manière dynamique, on peut ouvrir et ensuite refermer les colonnes avec la méthode ***setColumnCollpased()***. De manière similaire, il existe aussi les méthodes ***setColumnStetchable()*** et ***setColumnShrinkable()***.

Exemple




```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:stretchColumns="1" >

    <TableRow>
        <TextView
              android:text="@string/demande_URL" />
        <EditText
              android:id="@+id/saisie"
              android:hint="@string/indication_URL"
              android:layout_span="3" />
    </TableRow>

    <View
         android:layout_height="2dip"
         android:background="#0000FF"
         android:layout_marginTop="10dip"
         android:layout_marginBottom="10dip" />
```

```
    <TableRow>
        <Button
            android:id="@+id/annuler"
            android:layout_column="2"
            android:text="@string/bouton_annuler" />
        <Button
            android:id="@+id/ok"
            android:text="@string/bouton_OK" />
     </TableRow>

</TableLayout>
```

E) ScrollView

Pour présenter beaucoup d'informations sur un espace réduit, une solution consiste à utiliser le défilement, afin que seule une partie des informations soient visibles à un instant donné. Les autres informations seront accessibles en faisant défiler l'écran vers le haut ou vers le bas. **`ScrollView`** est le conteneur permettant de gérer le défilement de l'écran. Il ne doit contenir qu'un seul élément fils. Celui-ci sera généralement un autre *layout*.

Android 1.5 a ajouté **`HorizontalScrollView`** qui fonctionne comme **`ScrollView`** mais en horizontal (utile par exemple dans un formulaire si les zones de saisie sont très longues).

Remarques :

- ✓ il n'est pas possible de gérer un défilement à la fois horizontal et vertical.
- ✓ Il faut pas utiliser une *ListView* et un *ScrollView* conjointement, car cette utilisation engendre des conflits entre les 2 défilements.

Schéma général

```
<ScrollView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" >

    <TableLayout>
        . . .
    </TableLayout>

</ScrollView>
```

F) GridLayout

Introduit par Android 4, il permet de placer les éléments fils dans une grille de lignes divisées en cellules (ou autrement dit en colonne). Son avantage est une grande précision dans le placement des *widgets*, notamment lorsqu'un alignement vertical et aussi horizontal s'imposent. Cette précision est liée au fait que l'on peut définir un nombre illimité de cellules (pour les situations exigeant une précision extrême, on peut indiquer des milliers de cellules par exemple).

On a la possibilité de laisser des cellules vides.

Exemple général

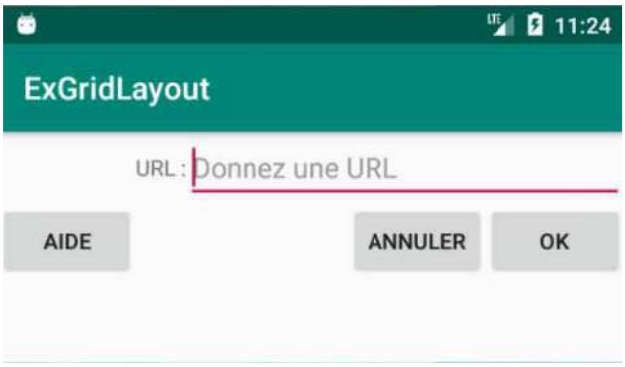
```
<GridLayout
        . . .
        android:columnCount= "2"
        android:rowCount="2">

        <Space
                android:layout_width="100dp"
                android:layout_column="0"
                android:layout_row="0" />

        <Button
                android:layout_column="1"
                android:layout_row="0"
                . . ./>

        <Button
                ndroid:layout_columnSpan="2"
                . . . />
</GridLayout>
```

On indique d'abord que la grille comportera 2 lignes et 2 colonnes.

Sur la première ligne (numéro 0), on trouvera un élément *Space* (donc de l'espace vide). Celui-ci occupera la première colonne d'une largeur de 100 dp. Ensuite, en colonne 1, il y aura un bouton.

Un deuxième bouton occupera la deuxième ligne. Son numéro de ligne n'est pas spécifié, mais par défaut, il se place sur la ligne suivante. Le bouton doit occuper les 2 colonnes (`layout_columnSpan="2"`).

Autre exemple




```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:columnCount="5"
    android:rowCount="2"
    tools:context=".MainActivity">

    <!-- Ligne 0 (première ligne)  -->
    <!-- ========================  -->
```

```
    <!-- pour indiquer que la première cellule de la ligne doit
         être vide, on utilise un élément Space
         Automatiquement cet élément occupera la colonne 0
         de la ligne 0 -->
    <Space
        />

    <!-- automatiquement le TextView va être placé en colonne 1,
         et ligne 0 -->
    <TextView
        android:layout_gravity="center_vertical"
        android:text="@string/demande_URL" />

    <!-- l'EditText se place automatiquement en colonne 2,
         et il doit occuper tout l'espace disponible en largeur,
         sur 3 colonnes -->
    <EditText
        android:id="@+id/saisie"
        android:hint="@string/indication_URL"
        android:layout_gravity="fill_horizontal"
        android:layout_columnSpan="3" />


    <!-- Ligne 1 (deuxième ligne)  -->
    <!-- ========================  -->

    <Button
        android:id="@+id/aide"
        android:layout_row="1"
        android:layout_column="0"
        android:text="@string/bouton_aide" />

    <!-- cet élément Space occupera l'espace sous le
         TextView contenant le texte "URL :" -->
    <Space
        />

    <!-- cet élément Space occupera tout l'espace
         laissé par les autres widgets sur la ligne -->
    <Space
         android:layout_gravity="fill_horizontal"/>

    <!-- l'indication de la colonne peut être supprimée -->
    <Button
        android:id="@+id/annuler"
        android:layout_column="3"
        android:text="@string/bouton_annuler" />

    <!-- l'indication de la colonne peut être supprimée -->
    <Button
        android:id="@+id/ok"
        android:layout_column="4"
        android:text="@string/bouton_OK" />
</GridLayout>
```

G) Remarques

Nous avons dit qu'il était possible d'imbriquer des *layouts* les uns dans les autres. Il faut savoir que cette configuration s'accompagne d'une perte d'efficacité (importante si les *widgets* de l'interface sont très nombreux). Il faut donc éviter les imbrications inutiles, et dans tous les cas de ne pas dépasser 10 niveaux.

La désérialisation de *layouts* est un processus coûteux, d'autant plus si des *layouts* sont imbriqués les uns dans les autres. Il est donc conseillé d'éviter de désérialiser un *layout* entier pour quelques changements mineurs dans un autre.

Pour nous aider à optimiser les hiérarchies de *layouts*, le SDK inclut un outil puissant, ***lint.*** Il détecte les problèmes de performances dans les *layouts*.

V) Les toasts et les alertes

Pour alerter l'utilisateur, il est possible d'utiliser d'autres mécanismes que les activités classiques :
- ✓ les notifications (voir plus loin)
- ✓ les toasts
- ✓ les alertes

Un **toast** est un message transitoire. Il s'affiche et disparaît de lui-même, sans intervention de l'utilisateur. Il ne modifie pas le focus de l'application courante : l'utilisateur peut continuer à interagir avec elle. La durée de l'affichage du toast est limité. Il est conçu pour diffuser des messages d'avertissement : une longue tâche en cous d'exécution, une batterie faible ...

Cependant, le programme ne peut pas savoir si l'utilisateur a pris connaissance ou pas du message diffusé par le toast.

La classe `Toast` contient une méthode statique `makeText` qui prend 3 paramètres :
- ✓ l'activité ou tout autre contexte qui a donné naissance au toast
- ✓ une chaîne de caractères ou un identifiant d'une ressource chaîne de caractères qui correspond au message affiché dans la boîte associée au toast
- ✓ une durée souhaitée pour l'affichage du message. En fait, l'une des 2 constantes `LENGTH_SHORT` ou `LENGTH_LONG`. Ces 2 constantes sont définies dans la classe `Toast.`

Lorsque le toast est configuré grâce à l'appel de la méthode `makeText,` il faut demander son affichage en invoquant la méthode `show().`

Une **alerte** peut être affichée avec une boîte de dialogue de type `AlertDialog`. Cette boîte est modale par rapport à l'activité qui lui a donné naissance. Elle reste affichée jusqu'à ce que l'utilisateur la ferme.

Pour créer une boîte de dialogue, on peut utiliser la classe `Builder` qui propose un ensemble de méthodes pour configurer une boîte de type `AlertDialog`. Chacune de ces méthodes renvoie une instance

de type `Builder` ce qui permet d'effectuer un chaînage des appels. En dernier lieu, il faut appliquer la méthode `show()` sur la dernière instance renvoyée pour afficher la boîte.

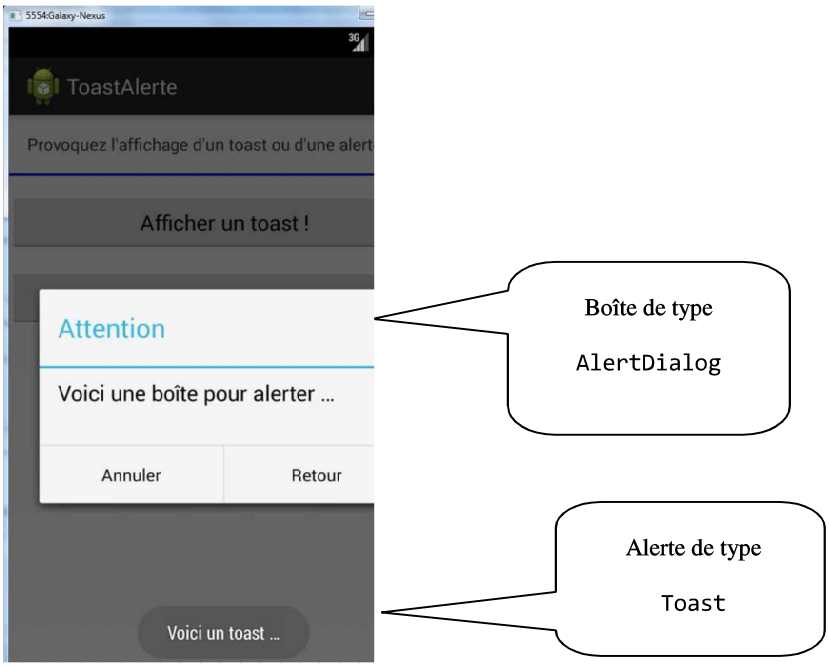
Voici les différentes méthodes à invoquer :

- ✓ le constructeur qui attend en paramètre le contexte (l'activité courante la plupart du temps)
- ✓ la méthode *`setTitle`* qui attend en paramètre la chaîne de caractères contenant le titre de la boîte de dialogue
- ✓ la méthode *`setMessage`* qui attend en paramètre le texte à afficher dans le corps de la boîte (une chaîne de caractères ou un identifiant d'une ressource chaîne de caractères)
- ✓ les méthodes *`setPositiveButton`*, *`setNeutralButton`*, et *`setNegativeButton`* permettant d'indiquer les boutons qui apparaîtront en bas de la boîte. Leur emplacement respectif sera à gauche, au centre et à droite. Chaque méthode a deux paramètres : le texte affiché sur le bouton, et le code qui sera invoqué lors du clic sur le bouton. Dans tous les cas, après un clic, la boîte se fermera automatiquement. Si l'on ne souhaite pas effectuer de traitement en particulier, on donnera la valeur `null`.

Exemple

L'exemple ci-après illustre l'utilisation des classes `Toast` et `AlertDialog`. L'activité principale est dotée de 2 boutons : un clic sur le premier doit provoquer l'affichage d'un toast, un clic sur le deuxième doit provoquer l'apparition d'une boîte d'alerte.

La copie d'écran ci-dessous montre le résultat quand les deux composants sont affichés simultanément.



```java
/*
 * Exemple d'utilisation des classes Toast et AlertDialog
 * ActiviteToastAlerte.java                                                    12/16
 */
package com.example.toastalerte;

import  . . .

/**
 * Cette activité affiche 2 boutons.
 * Un clic sur le premier provoque l'affichage d'un toast.
 * Un clic sur le deuxième provoque l'affichage d'une boîte d'alerte dotée
 * de 2 boutons.
 * @author LP MMS
 * @version 1.0
 */
public class ActiviteToastAlerte extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activite_toast_alerte);
    }

    /**
     * Méthode exécutée automatiquement lorsque l'on clique sur le bouton
     * "afficher un toast"
     * @param bouton  bouton sur lequel l'utilisateur a cliqué
     */
    public void clicSurToast(View bouton) {
        Toast.makeText(this, R.string.messageToast, Toast.LENGTH_LONG)
             .show();
    }

    /**
     * Méthode exécutée automatiquement lorsque l'on clique sur le bouton
     * "afficher une alerte"
     * @param bouton  bouton sur lequel l'utilisateur a cliqué
     */
    public void clicSurAlerte(View bouton) {
        new AlertDialog.Builder(this)
                    .setTitle(R.string.titreAlerte)
                    .setMessage(R.string.messageAlerte)
                    .setNeutralButton(R.string.texteAnnuler, null)
                    .setPositiveButton(R.string.texteRetour, null)
                    .show();
    }
}
```

VI) DatePicker

`DatePicker` est un *widget* permettant de sélectionner une date. La sélection de la date peut se faire via un *spinner* (une liste tournante) pour choisir le jour, le mois et l'année ou via un calendrier, ou bien les deux. Le programmeur peut fixer une date minimale et une date maximale parmi celles possibles.

Exemple de `DatePicker` dans lequel la sélection se fait avec 3 *spinners*.

```
<DatePicker
      android:id="@+id/selecteurDate"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:calendarViewShown="false"
      android:spinnersShown="true" />
```

Dans le code Java, on peut ensuite consulter les 3 valeurs choisies par l'utilisateur, grâce à 3 méthodes de la classe `DatePicker` :

`getDayOfMonth()`	renvoie le jour saisi
`getMonth()`	renvoie le mois saisi (Attention : un entier entre **0 et 11** !)
`getYear()`	renvoie l'année saisie

Autres composants utiles

- Sur le même principe que `DatePicker`, il existe une classe **`TimePicker`** pour sélectionner un horaire en heures et minutes.
- La classe **`DatePickerDialog`** correspond à une boîte de dialogue permettant d'effectuer la saisie d'une date.
- Pour visualiser un calendrier, on utilise **`CalendarView`**.
- Pour afficher une liste de choix dans une fenêtre *popup* , on utilise **`ListPopupWindow`**.
- Pour afficher un sélecteur de nombre, on utilise **`NumberPicker`**.

Les interfaces utilisateur - Compléments
Les listes - Les onglets - Les menus - ToolBar

I) Widgets de sélection - Les listes – Utilisation de ListView

Une liste est un ensemble d'éléments s'affichant les uns à la suite des autres. Pour sa présentation, chaque élément peut occuper de une à trois lignes. Et chaque ligne peut contenir plusieurs *widgets* : zone de texte, bouton, image …

Pour gérer les données à afficher dans une liste, il faut utiliser à un **adaptateur**. Un adaptateur joue le rôle d'intermédiaire entre les données à afficher et la vue qui s'occupera de leur présentation. Il se présente sous la forme d'une classe Java.

1.1) Démarche générale pour créer et afficher une liste de chaînes de caractères

1) Définir la liste des chaînes de caractères de la liste soit en dur dans le code Java (tableau, *ArrayList* ou toute autre collection), ou mieux dans un fichier ***XML*** de ressources, ou encore à partir d'une base de données

2) Dans le fichier ***XML*** décrivant la vue, il faut prévoir un élément de type ***ListView*** pour afficher la liste

3) La classe Java gérant l'activité de la vue peut éventuellement hériter de ***ListActivity*** au lieu de ***Activity***. Cette possibilité conduit à une classe activité légèrement plus simple, mais n'est possible que si l'activité gère une seule liste

4) Si la liste des chaînes de caractères a été définie dans un fichier ***XML*** de ressources, il faut récupérer cette liste dans le code Java en faisant un appel à la méthode ***getResources.*** Les valeurs récupérées sont stockées dans un tableau ou une *ArrayList*, par exemple.

5) Déclarer dans la classe activité, un attribut de type adaptateur, par exemple ***ArrayAdapter***

6) Créer et initialiser cet adaptateur avec la liste des valeurs à afficher. Il s'agit des valeurs placées dans un tableau ou une ***ArrayList***, comme évoqué à l'étape 4.

7) Associer cet adaptateur au *widget* de type ***ListView*** chargé de l'affichage de la liste :
   - ✓ Pour ce faire, on invoque la méthode ***setAdapter*** à laquelle on donne en argument l'adaptateur. Cette méthode doit être appliquée au *widget* de type ***ListView***.
   - ✓ Si par contre, la classe activité hérite de ***ListActivity***, on invoque la méthode ***setListAdapter*** sur l'activité courante.

Une fois la liste créé et affichée, il restera à gérer les interactions de la part de l'utilisateur. Cette gestion est expliquée dans les paragraphes suivants.

1.2) Comment créer et initialiser un adaptateur ?

Supposons que l'on souhaite afficher un ensemble de pays dans une liste. L'adaptateur le plus simple est **`ArrayAdapter`**. Son constructeur attend en argument un tableau ou une instance de *List*. Pour créer l'adaptateur, nous procèderons ainsi :

```
String[] lesPays = { "France", "Italie", "Espagne", "Portugal", "Allemagne"};

// On crée un adaptateur pour rassembler les données à afficher
ArrayAdapter<String> adaptateur =
                        new ArrayAdapter<String>(this,
                                                 android.R.layout.simple_list_item_1,
                                                 lesPays);
```

Notons que la classe ***ArrayAdapter*** est générique et paramétrée par le type des valeurs à afficher dans la liste.

Le constructeur de **`ArrayAdapter`** attend 3 paramètres :

- ✓ le contexte d'utilisation (généralement l'activité)
- ✓ l'identifiant de la ressource de la vue à utiliser pour afficher chacun des éléments de la liste. Ici, on a utilisé `android.R.layout.simple_list_item_1`. Il s'agit d'un *layout* prédéfini par Android (reconnaissable car son identifiant débute **`android.R`**) qui a la particularité de comporter un unique *widget* de type *TextView*
- ✓ le tableau ou la liste des éléments à afficher

Par défaut, ce constructeur appellera la méthode ***toString()*** sur les objets à afficher, puisqu'ils ne sont pas nécessairement des chaînes de caractères, et enveloppera chaque chaîne dans la vue donnée en tant que deuxième argument (c'est-à-dire dans `android.R.layout.simple_list_item_1`).

Il existe par exemple un autre identifiant de ressource prédéfini permettant d'afficher un élément de liste constitué cette fois de deux informations. Il s'agit de `android.R.layout.simple_list_item_2`. Cette vue pourrait être utilisée pour afficher, par exemple, en tant qu'élément de la liste : un numéro de téléphone sur une ligne, et un nom de personne sur la deuxième ligne.

Le *widget* classique pour afficher des listes est **`ListView`**. Une fois l'adaptateur créé et initialisé, il faudra encore :

- ✓ dans le fichier XML décrivant la vue inclure un élément **`ListView`**
- ✓ dans la classe Java décrivant l'activité, appeler sur le *widget* ***ListView*** la méthode ***setAdapter*** avec en argument un adaptateur pour fournir les données à afficher
- ✓ attacher un écouteur, grâce à la méthode ***setOnItemSelectedListener(),*** pour que l'activité soit informée lorsque l'utilisateur effectuera une sélection dans la liste.

1.3) Exemple d'une activité dérivant de *ListActivity*

Si l'activité comporte une seule liste, il sera plus simple de créer une activité spécialisée héritant de la classe **`ListActivity`** au lieu de **`Activity`**. La classe **`ListActivity`** hérite elle-même de **`Activity`**. Elle dispense le programmeur de gérer entièrement la **`ListView`** associée à l'activité.

De plus, si la vue principale est constituée uniquement d'une liste, il n'est pas nécessaire de fournir une vue pour l'activité. Par défaut, **`ListActivity`** construira une vue qui occupera tout l'écran.

Nous souhaitons afficher la liste des pays de l'UE et demander à l'utilisateur d'un choisir un. Les pays étant trop nombreux pour être tous affichés, un défilement sera ajouté automatiquement. Quand l'utilisateur cliquera sur l'un d'eux, celui-ci s'affichera dans un `TextView` situé au dessus de la liste.

1) Nous définissons d'abord les noms de pays dans un fichier de ressources ***XML***. Un tableau de chaînes de caractères peut être défini dans le fichier ***strings.XML***, de la manière suivante :

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

    <string name="app_name">ListePaysUE</string>
    <string name="Bienvenue">Bienvenue dans l\'UE ! Choisissez un pays parmi les </string>
    <string name="Choix">Vous avez choisi : </string>

    <string-array name="pays_UE">
        <item>Allemagne</item>
        <item>Autriche</item>
        <item>Belgique</item>
         ......
    </string-array>

</resources>
```

Fichier *strings.xml* - Avec un tableau de chaînes de caractères (extrait)

2) Nous définissons ensuite la vue dans un fichier ***XML*** de la manière suivante. Celle-ci comporte un élément de type ***ListView***. C'est lui qui permettra d'afficher la liste.

Remarque importante : comme nous prévoyons de faire hériter la classe activité de *ListActivity*, il faut obligatoirement que l'identifiant de l'élément `ListView` soit : *"@android:id/list"*

```
<!--  Fichier décrivant la vue associée à l'activité qui affiche les pays de l'Union
      Européenne
      Les pays sont affichés dans une liste, un widget de type ListView  -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="${relativePackage}.${activityClass}"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/LabelBienvenue"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dip"
        android:layout_marginBottom="10dip"
        android:text="@string/Bienvenue" />

     <TextView
        android:id="@+id/LabelChoix"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dip"
        android:layout_marginBottom="10dip"
        android:text="@string/Choix" />

     <!--  liste pour afficher les pays de l'Union Européenne
           REMARQUE : l'identifiant doit obligatoirement être "@android:id/list"
                      puisque la classe décrivant l'activité hérite de ListActivity -->
    <ListView
        android:id="@android:id/list"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:drawSelectorOnTop="false" />
</LinearLayout>
```

Fichier *activity_main.xml* décrivant la vue

3) Il faut maintenant écrire le code décrivant l'activité. Rappelons que la classe héritera de ***ListActivity***, au lieu de ***Activity***.

```
/*
 * Présente la liste des pays de l'union européenne
 * MainActivity.java                                    02/15
 */
package com.example.listepaysue;
import android.app.ListActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.TextView;
/**
 * Cette activité présente à l'utilisateur la liste des pays de l'Union Européenne.
 * Elle illustre l'utilisation d'un widget de type ListView et de la classe ListActivity
 * au lieu de Activity.
 * Lorsque l'utilisateur clique sur l'un des pays de la liste, celui-ci vient s'afficher
 * dans un TextView situé au dessus de la liste.
 * @author LP MMS
 * @version 1.0
 */
public class MainActivity extends ListActivity {

     /**
```

```
     * Label avec message de bienvenue
     */
    private TextView labelBienvenue;

    /**
     * Label sur lequel s'affiche le pays choisi par l'utilisateur
     */
    private TextView labelChoix;

    /**
     * Liste des pays présentés par l'application
     */
    private  String[] lesPays;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // on récupère les références sur les TextView
        labelBienvenue = (TextView) findViewById(R.id.LabelBienvenue);
        labelChoix = (TextView) findViewById(R.id.LabelChoix);

        // on accède aux noms des pays définis en tant que ressource
        lesPays = getResources().getStringArray(R.array.pays_UE);

        // On crée un adaptateur pour rassembler les données à afficher
        ArrayAdapter<String> adaptateur =
                new ArrayAdapter<String>(this,
                                         android.R.layout.simple_list_item_1,
                                         lesPays);
        setListAdapter(adaptateur);

        // on adapte le message du label de bienvenue (indiquer le nombre de pays)
        labelBienvenue.append(" " + lesPays.length);
    }

    /**
     * Méthode invoquée automatiquement si l'utilisateur sélectionne un élément
     * de la liste (méthode héritée de ListActivity)
     * @param parent     Element ListView dans lequel le clic s'est produit
     * @param v            La vue sur laquelle l'utilisateur a cliqué (un élément de la liste)
     * @param position      La position de cette vue dans la liste
     * @param id         L'identifiant de l'élément sur lequel le clic s'est
     *                   produit
     */
    public void onListItemClick(ListView parent, View v, int position, long id) {

        // on affiche le nom du pays sélectionné dans le label
        labelChoix.setText(R.string.Choix);
        labelChoix.append(" " + lesPays[position]);
    }
}
```

Classe activité

Remarque

La classe ***ListActivity*** contient une méthode ***getListView*** qui renvoie une référence sur l'objet de type *ListView* géré par l'activité.

1.4) Même exemple avec une activité qui hérite de *Activity*

a) Dans le fichier XML décrivant la vue, le *widget* ***ListView*** possède un identifiant défini de manière classique :

```
<!--  liste pour afficher les pays de l'Union Européenne
      REMARQUE : dans cette version, l'identifiant est choisi librement
                 par le programmeur    -->
<ListView
    android:id="@+id/listePays"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:drawSelectorOnTop="false" />
```

b) Dans la classe activité, il faut noter les changements suivants :

- la classe hérite de ***Activity*** et pas de *ListActivity*
- la classe implémente l'interface *OnItemClickListener* car nous avons décidé qu'elle serait son propre écouteur de l'événement clic sur la liste (il y a des variantes possibles, tout comme expliqué pour la gestion d'un clic sur un bouton dans le chapitre précédent)
- un attribut de type ***ListView***, nommé *listeDesPays*, a été défini et initialisé avec l'élément *ListView* du fichier XML
- l'adaptateur est associé à l'attribut *listeDesPays*
- un écouteur est associé à l'attribut *listeDesPays*
- comme la classe est son propre écouteur, on a défini la méthode ***onItemClick*** qui sera invoquée automatiquement lors du clic sur un élément de la liste

```
/*
 * Présente la liste des pays de l'union européenne
 * MainActivity.java                                    11/19
 */

package pays.gestionliste.exemple.com;
import . . .
/**
 * Cette activité présente à l'utilisateur la liste des pays de l'Union Européenne.
 * Elle illustre l'utilisation d'un widget de type ListView.
 * Dans cette version, la classe activité hérite de Activiy (et pas de ListActivity)
 * Lorsque l'utilisateur clique sur l'un des pays de la liste, celui-ci vient s'afficher
 * dans un TextView situé au dessus de la liste.
 * La classe activité est son propre écouteur de l'événement "clic sur la liste" : elle
 * implémente donc l'interface OnItemClickListener et en conséquence définit la méthode
 * onItemClick (méthode de l'interface OnItemClickListener), méthode qui sera invoquée
 * automatiquement lorsque l'utilisateur cliquera sur un élément de la liste.
 * @author C. Servières
 * @version 1.0
 */
public class MainActivity extends AppCompatActivity
                                    implements AdapterView.OnItemClickListener {
    /**
     * Label avec message de bienvenue
     */
    private TextView labelBienvenue;

    /**
     * Label sur lequel s'affiche le pays choisi par l'utilisateur
     */
    private TextView labelChoix;
```

```
    /**
     * Widget qui affiche la liste des pays de l'UE
     */
    private ListView listeDesPays;

    /**
     * Liste des pays présentés par l'application
     */
    private  String[] lesPays;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // on récupère les références sur les Widgets
        labelBienvenue =  findViewById(R.id.LabelBienvenue);
        labelChoix =  findViewById(R.id.LabelChoix);
        listeDesPays = findViewById(R.id.listePays);

        // on accède aux noms des pays définis en tant que ressource
        lesPays = getResources().getStringArray(R.array.pays_UE);

        // On crée un adpateur pour rassembler les données à afficher
        ArrayAdapter<String> adaptateur =
                new ArrayAdapter<String>(this,
                        android.R.layout.simple_list_item_1,
                        lesPays);

        // l'adaptateur est associé au widget listeDesPays
        listeDesPays.setAdapter(adaptateur);

        // on associe un écouteur pour intercepter les clics sur la liste des pays
        listeDesPays.setOnItemClickListener(this);

        // on adapte le message du label de bienvenue (indiquer le nombre de pays)
        labelBienvenue.append(" " + lesPays.length);
    }

    /**
     * Méthode invoquée automatiquement si l'utilisateur sélectionne un élément
     * de la liste (méthode définie dans l'interface OnItemClickListener)
     * @param parent    Element ListView dans lequel le clic s'est produit
     * @param v         La vue sur laquelle l'utilisateur a cliqué (un élément de la liste)
     * @param position  La position de cette vue dans la liste
     * @param id        L'identifiant de l'élément sur lequel le clic s'est produit
     */
    @Override
    public void onItemClick(AdapterView<?> parent, View v, int position, long id) {

        // on affiche le nom du pays sélectionné dans le label
        labelChoix.setText(R.string.Choix);
        labelChoix.append(" " + lesPays[position]);
    }
}
```

1.5) Remarques à propos de l'interface *Adapter*

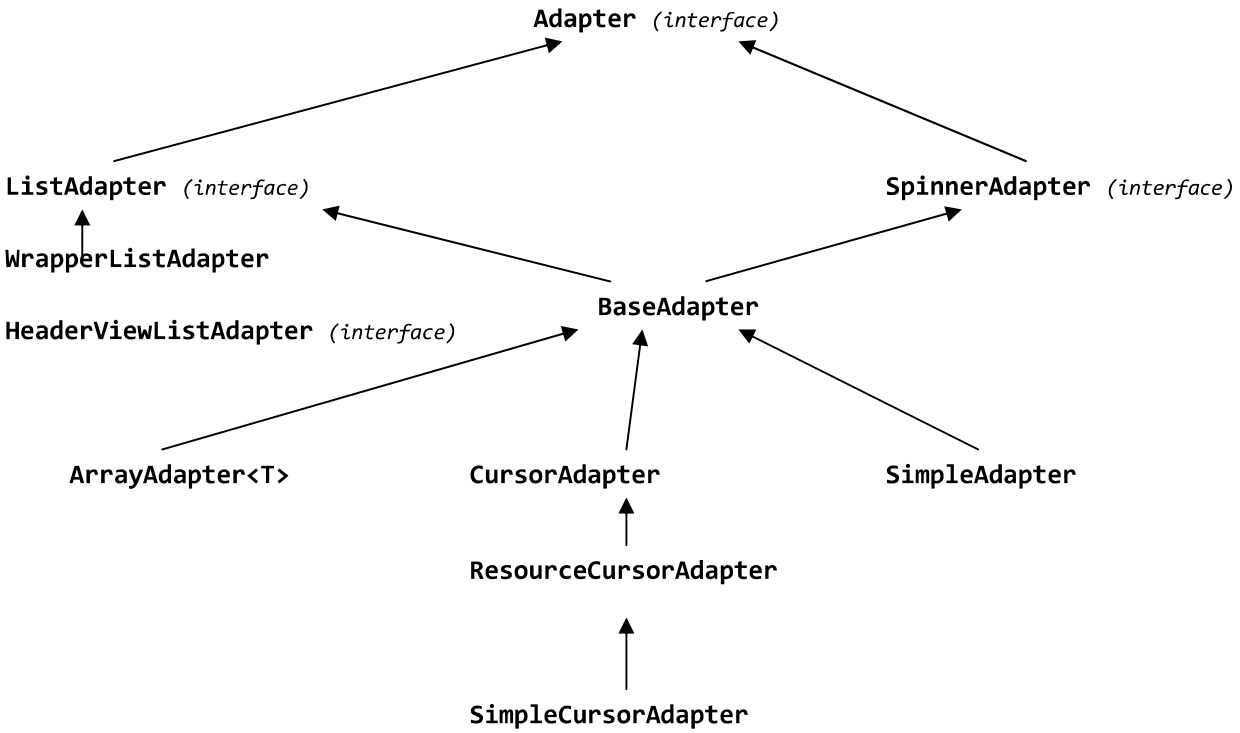
En résumé, pour afficher une liste d'éléments cliquables, nous avons besoin :

- ✓ de données placées dans un tableau, une *ArrayList*, ou toute autre collection
- ✓ d'un adaptateur qui jouera le rôle d'intermédiaire entre les données à afficher et la vue qui gèrera l'affichage
- ✓ d'une ***ListView*** qui assurera l'affichage

L'adaptateur est défini par le biais d'une classe Java qui implémente l'interface **`Adapter`**. La plateforme Android propose plusieurs classes prédéfinies qui implémentent celle-ci :

- ✓ **`ArrayAdapter<T>`** fonctionne à partir de tableaux ou assimilés, le plus utilisé
- ✓ **`BaseAdapter`** pour que le programmeur crée des adaptateurs personnalisés
- ✓ **`CursorAdapter`** pour traiter des données de type **`Cursor`**. Utile pour afficher des données en provenance d'une base de données à laquelle on accède via un curseur
- ✓ **`HeaderViewListAdapter`** pour ajouter des en-têtes et pieds de page aux **`ListView`**
- ✓ **`ResourceCursorAdapter`** pour la création de vues à partir d'une disposition XML
- ✓ **`SimpleAdapter`** pour afficher des données complexes, comme par exemple plusieurs valeurs dans un même élément de la liste
- ✓ **`SimpleCursorAdapter`** est une implémentation de l'interface **`CursorAdapter`** permettant de gérer facilement des données de type chaîne de caractères ou image

Le programmeur peut définir ses propres classes d'adaptateur. Il doit pour se faire définir une nouvelle classe qui hérite de `BaseAdapter` ou bien de l'une des classes adaptateurs déjà définies.



Hiérarchie d'héritage des principales classes ou interfaces *adaptateurs*

1.6) Méthodes Java pour modifier le contenu d'un adaptateur

Les méthodes ci-dessous sont communes aux différents adaptateurs et utiles si on veut modifier la liste des données affichées de manière dynamique, donc dans le code Java. Attention, ces méthodes ne fonctionnent que si les données liées à l'adaptateur sont stockées dans un objet *mutable* (donc modifiable), comme par exemple une *ArrayList*. Par contre, elles ne seront pas utilisables si les données sont stockées dans un tableau.

Les méthodes les plus courantes sont :

void add(T objet)	pour ajouter un objet à un adaptateur
void addAll(une collection)	pour ajouter tous les objets de la collection argument
void insert(T objet, int position)	pour ajouter un objet à une position précise
T getItem(int position)	pour récupérer l'objet situé à la position indiquée dans la liste
int getPosition(T object)	pour récupérer la position d'un objet précis
void remove(T object)	pour supprimer un objet de l'adaptateur
void clear()	pour vider complètement l'adaptateur
void notifyDataSetChanged()	pour informer l'adaptateur qu'un changement a eu lieu dans la collection qui lui est associée. Un appel à cette méthode permettra d'actualiser la vue en fonction du changement.

Il est possible aussi d'appliquer des méthodes sur l'instance de type *ListView*. On pourra invoquer, par exemple :

void clearChoices()	pour désélectionner les éléments préalablement sélectionnés
void requestLayout()	pour que la vue soit mise à jour (visuellement)

1.7) Comment sélectionner un ou des éléments dans une liste ?

Par défaut, ***ListView*** permet seulement de détecter les clics de l'utilisateur sur les éléments de la liste. Cependant, on aura parfois besoin qu'une liste mémorise un ou plusieurs choix de l'utilisateur. Chaque élément de la liste sera alors doté d'une case à cocher, située à droite. Pour ce faire, il faut intégrer des informations supplémentaires au *widget* ***ListView***.

a) Il faut changer le mode du *widget* ***ListView*** avec l'une ou l'autre des techniques suivantes :

- soit dans le fichier XML, en ajoutant la propriété :
  `android:choiceMode="multipleChoice"` ou
  `android:choiceMode="singleChoice"`
- soit dans le code Java, en invoquant la méthode ***setChoiceMode*** sur l'objet ***ListView*** à modifier. On donne en paramètre de la méthode la constante **CHOICE_MODE_SINGLE** ou **CHOICE_MODE_MULTIPLE**

b) Puis, on donne en paramètre du constructeur d'***ArrayAdapter***, la valeur
`android.R.layout.simple_list_item_single_choice`, ou
`android.R.layout.simple_list_item_multiple_choice`
(au lieu de `android.R.layout.simple_list_item_1`)

c) Dans le code Java, on fait appel à l'une ou l'autre des méthodes suivantes :
SparseBooleanArray getCheckedItemPositions() qui renvoie les positions des éléments sélectionnés. ***SparseBooleanArray*** est une classe prédéfinie qui établit une correspondance entre des entiers et des booléens.

ou bien ***long[ ] getCheckedItemIds()*** qui renvoie les identifiants des éléments sélectionnés

Exemple d'utilisation de *getCheckedItemPositions*

On suppose que `listeProposition` est de type ***ListView***.

```
// pour obtenir les mots sélectionnés par l'utilisateur
// (sous la forme d'une table qui associe aux éléments cochés un booléen)
SparseBooleanArray elementCoche = listeProposition.getCheckedItemPositions();

// on parcourt la table des mots cochés et on les affiche sur la console
for (int i = 0; i < elementCoche.size(); i++) {

    if (elementCoche.valueAt(i)) {     // valueAt(i) renvoie vrai ssi l'élément
                                       // d'indice i de la table est coché
        /*
         * Le mot d'indice i de la table est coché
         * On accède au libellé de celui-ci et on l'affiche sur la console
         */
        System.out.print(
            listeProposition.getAdapter().getItem(elementCoche.keyAt(i)).toString());
    }
}
```

II) Widgets de sélection - Les listes – Utilisation de Spinner

2.1) Le *widget* Spinner

Ce *widget* est l'équivalent des listes déroulantes classiques. La différence avec une ***ListView*** est que la ***ListView*** occupe tout l'écran (ou presque).

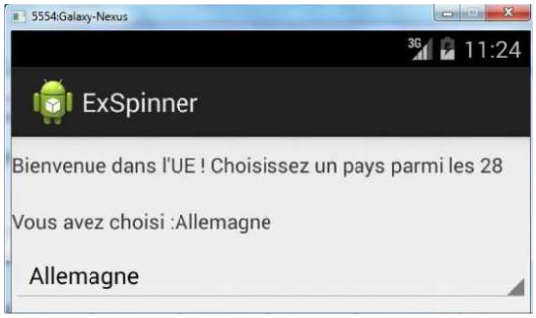
Un ***Spinner*** combine à la fois un ***TextView*** et une ***ListView***. Au lancement, le ***TextView*** affiche donc un seul élément de la liste. Il est doté d'un bouton permettant d'afficher les autres choix.

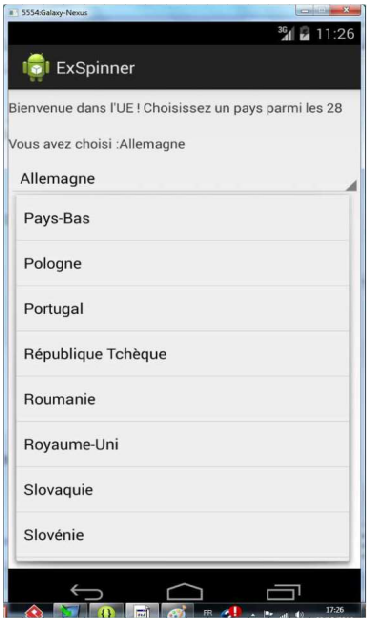
```
View
     ViewGroup                          (abstraite)
          AdapterView<T>                (abstraite)
               AbsListView              (abstraite)
                    ListView
               AbsSpinner               (abstraite)
                     Spinner
```

Hiérarchie d'héritage pour situer *ListView* et *Spinner*

2.2) Exemple d'utilisation de *Spinner*

Reprenons l'exemple précédent, en utilisant un *Spinner* à la place de *ListView*. Au lancement, on verra apparaître seulement le premier pays de la liste. Si l'utilisateur clique sur le triangle à droite, la liste apparaîtra éventuellement avec une barre de défilement.





Dans le fichier *XML* décrivant la vue, on utilise l'élément *Spinner* au lieu de *ListView*.

```
<!--  Fichier décrivant la vue associée à l'activité qui affiche les pays de l'Union Européenne
      Les pays sont affichés dans une liste, un widget de type Spinner  -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="${relativePackage}.${activityClass}"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/LabelBienvenue"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dip"
        android:layout_marginBottom="10dip"
        android:text="@string/Bienvenue" />

     <TextView
        android:id="@+id/LabelChoix"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dip"
        android:layout_marginBottom="10dip"
        android:text="@string/Choix" />

    <Spinner
        android:id="@+id/spinner"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

</LinearLayout>
```

Fichier *activity_main.xml* décrivant la vue

La classe Java décrivant l'activité hérite de ***Activity*** et implémente l'interface ***OnItemSelectedListener***. Comme dans l'exemple précédent, il faut créer un adaptateur qui servira d'intermédiaire entre les données de la liste et le *widget* de type *Spinner* chargé d'afficher la liste. L'adaptateur est associé à la liste de type *Spinner* en invoquant la méthode ***setAdapter***.

L'interface ***OnItemSelectedListener*** contient deux méthodes que la classe activité doit donc redéfinir :

- ✓ ***onItemSelected*** est invoquée automatiquement lorsqu'un item de la liste est sélectionné par l'utilisateur
- ✓ ***onNothingSelected*** est invoquée automatiquement si la liste est refermée alors qu'aucun élément de celle-ci n'a été sélectionné

```
/*
 * Présente la liste des pays de l'union européenne
 * MainActivity.java                                    02/15
 */
package com.example.exspinner;
import . . .

/**
 * Cette activité présente à l'utilisateur la liste des pays de l'Union Européenne.
 * Elle illustre l'utilisation d'un widget de type Spinner.
 * Lorsque l'utilisateur clique sur l'un des pays de la liste, celui-ci vient s'afficher
 * dans un TextView situé au dessus de la liste.
```

```
 * @author LP MMS
 * @version 1.0
 */
public class MainActivity extends Activity
                          implements OnItemSelectedListener {

    /**
     * Label avec message de bienvenue
     */
    private TextView labelBienvenue;

    /**
     * Label sur lequel s'affiche le pays choisi par l'utilisateur
     */
    private TextView labelChoix;

    /**
     * Liste des pays présentés par l'application
     */
    private  String[] lesPays;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // on récupère les références sur les TextView, et la liste Spinner
        labelBienvenue = (TextView) findViewById(R.id.LabelBienvenue);
        labelChoix = (TextView) findViewById(R.id.LabelChoix);

        Spinner liste = (Spinner) findViewById(R.id.spinner);

        // on associe un écouteur à la liste déroulante
        liste.setOnItemSelectedListener(this);

        // on accède aux noms des pays définis en tant que ressource
        lesPays = getResources().getStringArray(R.array.pays_UE);

        // On crée un adaptateur pour rassembler les données à afficher
        ArrayAdapter<String> adaptateur =
                new ArrayAdapter<String>(this,
                                         android.R.layout.simple_spinner_item,
                                         lesPays);
        adaptateur.setDropDownViewResource(
                                  android.R.layout.simple_spinner_dropdown_item);
        liste.setAdapter(adaptateur);

        // on adapte le message du label de bienvenue (indiquer le nombre de pays)
        labelBienvenue.append(" " + lesPays.length);
    }


    /**
     * Méthode invoquée automatiquement lorsqu'un élément de la liste est sélectionné
     * @param parent   l'AdapterView dans lequel la sélection a été effectuée
     * @param v               la vue qui est associée à cet AdapterView
     * @param position        la position de l'élément qui a été sélectionné
     * @param id              l'identifiant de l'élément qui a été sélectionné
     */
    public void onItemSelected(AdapterView<?> parent, View v, int position, long id) {
        labelChoix.append(lesPays[position]);
    }

    /**
     * Méthode invoquée automatiquement si la liste est refermée alors qu'aucun
     * élément de la liste n'a été sélectionné
     * @param parent   l'AdapterView dans lequel la sélection aurait pu avoir lieu
```

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

Classe activité

III) Pour aller plus loin avec les listes

3.1) Personnalisation d'une liste

Comme indiqué précédemment, le programmeur peut définir ces propres adaptateurs. Voici les étapes à suivre. Nous supposons que nous souhaitons afficher dans chaque « casier » de la liste 2 informations : la nature d'une dépense et le montant de celle-ci.

TOTAL	380
Achat carburant	80
Assurance	300
Contrôle technique	0
Vidange	0
Réparation	0
Autre	0

a) Créer une classe ***ItemDepense*** contenant 2 attributs, le premier de type *String* correspondant à la nature de la dépense, le deuxième de type *int* correspondant au montant. On définit également dans cette classe un constructeur avec arguments et des accesseurs.

b) Créer un fichier XML décrivant la vue de chaque « casier » de la liste. Pour l'exemple, on pourrait avoir à la racine un *LinearLayout* avec orientation horizontale, 2 *TextView* pour afficher les 2 informations. Supposons que ce fichier se nomme ***vue_item_liste.xml.***

```
<?xml version="1.0" encoding="utf-8"?>
<!--  Vue qui décrit chacune des lignes de la liste présentant le bilan
      des frais (dans l'activité ActiviteFraisVoiture)
      Chaque item de la liste contient 2 informations :
         - la nature de la dépense
         - son montant
      La largeur des TextView est égale à 0, le poids correspond donc à un pourcentage
      d'occupation de l'espace en largeur (respectivement 80% et  20%)
      fichier vue_item_liste.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal">

    <!--  pour afficher la nature du frais -->
    <TextView
        android:id="@+id/item_nature"
        android:layout_marginTop="@dimen/marge_haute"
        android:layout_marginBottom="@dimen/marge_basse"
        android:layout_marginLeft="@dimen/marge_gauche"
        android:layout_width="0dp"
        android:layout_height="fill_parent"
        android:layout_weight="80" />

    <!--  pour afficher le montant du frais -->
```

```
    <TextView
        android:id="@+id/item_montant"
        android:layout_marginTop="@dimen/marge_haute"
        android:layout_marginBottom="@dimen/marge_basse"
        android:layout_width="0dp"
        android:layout_height="fill_parent"
        android:layout_weight="20" />

</LinearLayout>
```

c) Définir une nouvelle classe adaptateur, par héritage de *ArrayAdapter* par exemple, si l'on suppose que le remplissage des données se fera via un tableau ou une liste.

```
/*
 * Adaptateur spécifique pour afficher le bilan des dépenses.
 * ItemDepenseAdapter.java
 */
package adaptateurpersonnalise.fraisvoiture.gestionliste.exemple.com;
import . . .

/**
 * Classe qui représente un adaptateur pour afficher une liste.
 * Plus précisément, cet adaptateur servira à afficher la liste contenant
 * un bilan des dépenses pour une année précise.
 * Les items de cette liste contiennent 2 informations : la nature de la
 * dépense et le montant (instance de type ItemDepense).
 * La classe hérite de ArrayAdapter. Rappel : la classe ArrayAdapter permet
 * de gérer une liste dont chaque item est constitué d'une seule valeur.
 * La classe ItemDepenseAdapter permettra quant à elle de gérer une liste
 * dont chaque item est consitué de 2 valeurs.
 * @author C. Servières
 * @version 1.0
 */
public class ItemDepenseAdapter extends ArrayAdapter<ItemDepense>{

   /** Identifiant de la vue permettant d'afficher chaque item de la liste */
   private int identifiantVueItem;

   /**
    * Constructeur de l'adaptateur
    * @param contexte    contexte de création de l'adaptateur
    * @param vueItem     identifiant de la vue permettant d'afficher chaque
    *                    item de la liste
    * @param lesItems    Liste des items à afficher
    */
   public ItemDepenseAdapter(Context contexte, int vueItem,
                             List<ItemDepense> lesItems) {
      super(contexte, vueItem, lesItems);
      this.identifiantVueItem = vueItem;
   }

   /**
    * Permet d'affecter à chaque item de la liste, les valeurs qui
    * doivent être affichées
     * @param position  position de l'élément qui doit être affiché
     *                  (position au sein de la liste associée à l'adaptateur)
     * @param uneVue     contient soit la valeur null, soit une ancienne vue
     *                   pour l'élément à afficher. La méthode pourra alors se
     *                   contenter de réactualiser cette vue
     * @param parent     vue parente à laquelle la vue à renvoyer peut être rattachée
     * @return une vue qui affichera les informations adéquates dans l'item de la liste
     *         situé à la position p
    */
   @Override
   public View getView(int position, View uneVue, ViewGroup parent) {

      // on récupère la valeur de l'item à afficher, via sa position
      ItemDepense uneDepense = getItem(position);
      LinearLayout vueItemListe; // layout décrivant un item de la liste

      if (uneVue == null) {

         /*
```

```
            * la vue décrivant chaque item de la liste n'est pas encore créée
            * Il faut désérialiser le layout correspondant à cette vue.
            */
           LayoutInflater outil;
           outil = (LayoutInflater)getContext()
                                .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
           vueItemListe = (LinearLayout) outil.inflate(identifiantVueItem,
                                                       parent, false);

        } else {
           vueItemListe = (LinearLayout) uneVue;
        }

        // on accède aux 2 widgets présents sur la vue
        TextView vueNature =  vueItemListe.findViewById(R.id.item_nature);
        TextView vueMontant =  vueItemListe.findViewById(R.id.item_montant);

        // on place dans les 2 widgets les valeurs de l'item à afficher
        vueNature.setText(uneDepense.getNature());
        vueMontant.setText(Integer.toString(uneDepense.getMontant()));
        return vueItemListe;
    }

}
```

La méthode à redéfinir est ***getView*** qui renvoie une vue permettant d'afficher *l'item* de la liste situé à la position argument, à partir de la source de données fixée lors de l'appel au constructeur (paramètre *lesItems* de celui-ci).

Le paramètre *uneVue* contient éventuellement une ancienne vue destinée à afficher un item de la liste. Il est égal à ***null*** si aucune vue n'est disponible. Dans un souci d'optimisation, le programmeur doit réutiliser la vue argument si elle existe (ce qui est fait dans le *else*) plutôt que d'en créer une nouvelle (ce qui est fait dans le *if*).

Pour instancier une vue, il faut utiliser un utilitaire de type ***LayoutInflater***. On récupère celui-ci en faisant appel à la méthode ***getSystemService*** et en donnant en argument la constante `Context.LAYOUT_INFLATER_SERVICE`. On applique alors sur celui-ci la méthode ***inflate*** en donnant en argument l'identifiant de la vue à créer (une référence au fichier *xml* contenant la vue, initialisé lors de l'appel au constructeur).

Optimisation

Il faut comprendre que la méthode *getView* sera invoquée très souvent. Il est donc intéressant de l'optimiser :
- L'appel à ***getSystemService*** est une opération coûteuse. On peut faire en sorte de l'invoquer une seule fois.
- Il est préférable également de ne pas appeler *findViewById* trop souvent. Grâce à la méthode ***setTag***, on peut sauvegarder les références des *TextView*, dans la vue elle-même. La méthode ***getTag*** permet ensuite de récupérer ces références.

On obtient alors la classe suivante :

```
public class ItemDepenseAdapterBis extends ArrayAdapter<ItemDepense>{

    /** Identifiant de la vue permettant d'afficher chaque item de la liste */
    private int identifiantVueItem;

    /**
     * Objet utilitaire permettant de désérialiser une vue
     */
```

```
    private LayoutInflater inflater;

    /** Regroupe les 2 TextView présents sur la vue d'un item de la liste */
    static class SauvegardeTextView {
        TextView natureDepense;
        TextView montantDepense;
    }

    /**
     * Constructeur de l'adaptateur
     * @param contexte    contexte de création de l'adaptateur
     * @param vueItem     identifiant de la vue permettant d'afficher chaque
     *                    item de la liste
     * @param lesItems    Liste de items à afficher
     */
    public ItemDepenseAdapterBis(Context contexte, int vueItem,
                                 List<ItemDepense> lesItems) {
        super(contexte, vueItem, lesItems);
        this.identifiantVueItem = vueItem;
        inflater = (LayoutInflater)getContext()
                .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
    }

    /**
     * Permet d'affecter à chaque item de la liste, les valeurs qui
     * doivent être affichées
     * @param position  position de l'élément qui doit être affiché
     *                  (position au sein de la liste associée à l'adaptateur)
     * @param uneVue     contient soit la valeur null, soit une ancienne vue
     *                   pour l'élément à afficher. La méthode pourra alors se
     *                   contenter de réactualiser cette vue
     * @param parent     vue parente à laquelle la vue à renvoyer peut être rattachée
     * @return une vue qui affichera les informations adéquates dans l'item de la liste
     *         situé à la position p
     */
    @Override
    public View getView(int position, View uneVue, ViewGroup parent) {

        // on récupère la valeur de l'item à afficher, via sa position
        ItemDepense ligneBilan = getItem(position);
        LinearLayout vueItemListe; // layout décrivant un item de la liste
        SauvegardeTextView sauve;   // regroupe les 2 TextView présent sur la vue
                                    // destinée à afficher l'item

        if (uneVue == null) {

            /*
             * la vue décrivant chaque item de la liste n'est pas encore créée
             * Il faut désérialiser le layout correspondant à cette vue.
             */
            uneVue = inflater.inflate(identifiantVueItem, parent, false);

            // on récupère un accès sur les 2 TextView qu'il faudra renseigner
            sauve = new SauvegardeTextView();
            sauve.natureDepense = uneVue.findViewById(R.id.item_nature);
            sauve.montantDepense = uneVue.findViewById(R.id.item_montant);

            // on stocke les identifiants de 2 TextView dans la vue elle-même
            uneVue.setTag(sauve);

        } else {

            // on récupère les identifiants des 2 TextView stockés dans la vue
            sauve = (SauvegardeTextView) uneVue.getTag();
        }

        // on place dans les 2 TextView les valeurs de l'item à afficher
        sauve.natureDepense.setText(ligneBilan.getNature());
        sauve.montantDepense.setText(Integer.toString(ligneBilan.getMontant()));
        return uneVue;
    }
}
```

3.2) *CardView* et *RecyclerView* depuis Android 5.0

Les ***CardView*** permettent d'afficher des informations dans des cartes dont il possible de personnaliser l'apparence. Cette classe hérite de *FrameLayout* et peut être utilisée conjointement avec *RecyclerView* ou *ListView*. L'affichage des ***CardView*** est conforme au design nommé *Material Design*. Une conséquence est l'ajout de la notion de profondeur dans l'affichage des cartes, ce qui se nomme ***elevation*** dans la terminologie Android.

Le composant ***RecyclerView*** permet de gérer une liste de manière plus flexible et plus avancée qu'une simple *ListView*. Il est pertinent de l'utiliser dès lors que les informations contenues dans chaque « casier » de la liste sont nombreuses. C'est une alternative à la personnalisation de l'adaptateur évoquée dans la section précédente.

En effet, avec un ***RecyclerView***, on peut utiliser un gestionnaire de mise en forme (*layout*) pour décrire le positionnement des *widgets* au sein des « casiers » de la liste. On peut aussi définir des animations précises pour les opérations courantes que l'on applique généralement à une liste, telles que suppression, ajout d'un élément par exemple.

Mais le principal avantage du ***RecyclerView*** est son efficacité car, lorsque l'utilisateur effectue un défilement sur la liste, il réutilise (recycle) les vues affichant les *items* de la liste au lieu de les recréer. Le gain d'efficacité porte sur le temps d'exécution et aussi sur l'occupation mémoire.

Pour intégrer à notre application un ***RecyclerView*** gérant une liste formée de ***CardView***, la démarche est la suivante :

- placer dans le fichier ***build.gradle*** les dépendances vers ces composants
```
implementation 'androidx.cardview:cardview:1.0.0'
implementation 'androidx.recyclerview:recyclerview:1.0.0'
```
- définir un *layout* avec comme élément racine un ***CardView*** pour afficher chaque item de la liste
- définir une classe ayant comme attributs toutes les informations affichées dans chaque *item* de la liste
- définir une classe dérivée de ***RecyclerView.ViewHolder*** pour faire lien entre une instance de la classe précédente et la vue permettant d'afficher un *item* de la liste
- définir un adaptateur spécialisé pour l'affichage des *items* de la liste. Il s'agit d'une classe dérivée de ***RecyclerView.Adapter*** par exemple

Exemple

On souhaite afficher une liste de photos de Paris avec pour chacune d'elles un libellé.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!-- Vue décrivant la disposition au sein de chaque item de la liste
     qui affiche les photos de Paris.
     Illustration de l'utilisation d'un CardView pour chaque item de
     la liste  -
     fichier vue_item_liste.xml                -->
<androidx.cardview.widget.CardView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="5dp">

    app:cardBackgroundColor="@android:color/white"
    app:cardCornerRadius="2dp"
    app:cardElevation="2dp"   >

    <!-- Les CardView possèdent des attributs supplémentaires dont
         - cardBackgroundColor
         - cardElevation pour l'élévation (donc aussi l'ombre)
         - cardCornerRadius pour arrondir les angles
     -->
    <!-- Le CardView présentera : une image, suivie d'un texte -->
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical">

        <ImageView
            android:id="@+id/image"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:scaleType="centerCrop"  />

        <TextView
            android:id="@+id/text"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:background="?android:selectableItemBackground"
            android:padding="20dp"
            android:fontFamily="sans-serif"
            android:textColor="#333"
            android:textSize="18sp" />
    </LinearLayout>

</androidx.cardview.widget.CardView>
```

vue_item_liste.xml

```java
public class PhotoParis {

    /** Libellé de la photo */
    private String libelle;

    /** Identifiant de la photo, au sein des ressources de type Drawable */
    private int photo;


    /**
     * Constructeur avec en argument les valeurs des 2 attributs
     * @param libelle     libellé de la photo
     * @param photo       identifiant de la photo
     */
    public PhotoParis(String libelle, int photo) {
        this.libelle = libelle;
        this.photo = photo;
    }

    /**
     * Renvoie le libellé de la photo
     * @return  une chaîne contenant le libellé
     */
    public String getLibelle() {
```

```
        return libelle;
    }

    /**
     * Renvoie l'identifiant de la photo
     * @return  un entier contenant l'identifiant de la photo
     */
    public int getPhoto() {
        return photo;
    }
}
```

PhotoParis.java

```
/**
 * Description du contenant des items de la liste de type RecyclerView
 * permettant d'afficher des photos de Paris.
 * @author C. Servières
 * @version 1.0
 */
public class PhotoViewHolder extends RecyclerView.ViewHolder{

    /**
     * TextView qui contient le libellé de la photo
     */
    private TextView libellePhoto;

    /**
     * ImageView qui contient la photo
     */
    private ImageView laPhoto;


    /**
     * Constructeur avec en argument une vue correspondant
     * à un item de la liste
     * Le constructeur permet d'initialiser les identifiants des
     * widgets déclarés en tant qu'attributs
     * @param itemView   vue décrivant l'affichage d'un item de la liste
     */
    public PhotoViewHolder(View itemView) {
        super(itemView);
        libellePhoto = (TextView) itemView.findViewById(R.id.text);
        laPhoto = (ImageView) itemView.findViewById(R.id.image);
    }

    /**
     * Permet de placer les informations contenues dans l'argument
     * dans les widgets d'un item de la liste
     * @param maPhoto   l'instance qui doit être affichée
     */
    public void bind(PhotoParis maPhoto){
        libellePhoto.setText(maPhoto.getLibelle());
        laPhoto.setImageResource(maPhoto.getPhoto());
    }
}
```

PhotoViewHolder.java

```
/**
 * Adaptateur spécifique pour afficher une liste de type RecyclerView
 * dont les items sont de type PhotoParis
 * @author Servières
 * @version 1.0
 */
public class PhotoAdapter extends RecyclerView.Adapter<PhotoViewHolder> {

    /**
```

```
     * Source de données à afficher par la liste
     */
    private List<PhotoParis> lesDonnees;

    /**
     * Constructeur avec en argument la liste source des données
     * @param donnees       liste contenant les instances de type
     *                      PhotoParis que l'adapteur sera chargé de gérer
     */
    public PhotoAdapter(List<PhotoParis> donnees) {
        lesDonnees = donnees;
    }

    /**
     * Renvoie un contenant de type PhotoViewHolder qui permettra d'afficher
     * un élément de la liste
     */
    @Override
    public PhotoViewHolder onCreateViewHolder(ViewGroup viewGroup, int itemType) {

        View view = LayoutInflater.from(
                viewGroup.getContext()).inflate(R.layout.vue_item_liste,
                                                viewGroup,false);
        return new PhotoViewHolder(view);
    }

    /**
     * On remplit un item de la liste en fonction de sa position
     */
    @Override
    public void onBindViewHolder(PhotoViewHolder myViewHolder, int position) {
        PhotoParis myObject = lesDonnees.get(position);
        myViewHolder.bind(myObject);
    }

    /**
     * Renvoie la taille de la liste
     */
    @Override
    public int getItemCount() {
        return lesDonnees.size();
    }
}
```

PhotoAdapter.java

```
/**
 * Activité qui affiche une liste de type RecyclerView.
 * Chaque item de la liste est défini via un CardView et contient
 * une photo de Paris et son libellé
 * @author Servières
 * @version 1.0
 */
public class MainActivity extends Activity {

    /**
     * Liste source des données à afficher :
     * chaque élément contient une instance de PhotoParis (une photo
     * et son libellé)
     */
    private ArrayList<PhotoParis> listePhoto;

    /**
     * Element permettant d'afficher la liste des photos
     */
    private RecyclerView photoRecyclerView;


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
```

```
        photoRecyclerView = findViewById(R.id.my_recycler_view);
        initialiseListePhoto();

        /*
         * On crée un gestionnaire de layout linéaire, et on l'associe à la
         * liste de type RecyclerView
         */
        LinearLayoutManager gestionnaireLineaire = new LinearLayoutManager(this);
        photoRecyclerView.setLayoutManager(gestionnaireLineaire);

        /*
         * On crée un adaptateur personnalisé et permettant de gérer spécifiquement
         * l'affichage des instances de type PhotoParis en tant que item de la liste.
         * Cet adapatateur est associé au RecyclerView
         */
        PhotoAdapter adaptateur = new PhotoAdapter(listePhoto);
        photoRecyclerView.setAdapter(adaptateur);
    }


    /**
     * Méthode pour initialiser la liste des photos et des textes
     */
    private void initialiseListePhoto() {
        listePhoto = new ArrayList<>();
        listePhoto.add(new PhotoParis("Bibliothèque BnF", R.drawable.bnf));
        listePhoto.add(new PhotoParis("Jardins du Luxembourg", R.drawable.luxembourg));
        listePhoto.add(new PhotoParis("Notre-Dame de Paris", R.drawable.notredame));
        listePhoto.add(new PhotoParis("La Seine", R.drawable.seine));
        listePhoto.add(new PhotoParis("La Tour Eiffel", R.drawable.toureiffel));
        listePhoto.add(new PhotoParis("La pont Alexandre III", R.drawable.pontalexande));
    }
}
```

MainActivity.java

IV) Les onglets

Selon la philosophie d'Android, les activités et les vues associées doivent être concises et claires. Lorsqu'une activité comporte trop d'informations, il est préférable de la décomposer en plusieurs activités. Un compromis satisfaisant consistera parfois à placer plusieurs onglets sur la vue que doit gérer l'activité. Ce compromis simplifie la tâche du programmeur qui gère ainsi une seule activité, et ne se préoccupe pas de la communication entre les activités.

Android dispose d'un conteneur nommé **`TabHost`**. Le principe est le suivant : une portion de ce qu'affiche l'activité (souvent la totalité) est liée à des onglets. Lorsque l'utilisateur clique sur un onglet, une autre partie de la vue s'affiche : celle associée à l'onglet cliqué.

Selon la terminologie Android, un onglet est constitué de 2 parties :

- ✓ un bouton d'onglet : la partie supérieure sur laquelle on peut cliquer pour rendre l'onglet visible
- ✓ le contenu de l'onglet

Dans le fichier XML décrivant la vue, nous utiliserons les éléments suivants :

- ✓ un conteneur **`TabHost`** qui contiendra la description des boutons d'onglet et le contenu des onglets
- ✓ un *widget* **`TabWidget`** qui représentera la ligne des boutons des onglets. Il contiendra les labels des boutons, et éventuellement des icônes
- ✓ un conteneur **`FrameLayout`** pour réunir le contenu des onglets. Généralement chaque contenu sera décrit par un *layout* qui lui-même contiendra des *widgets*

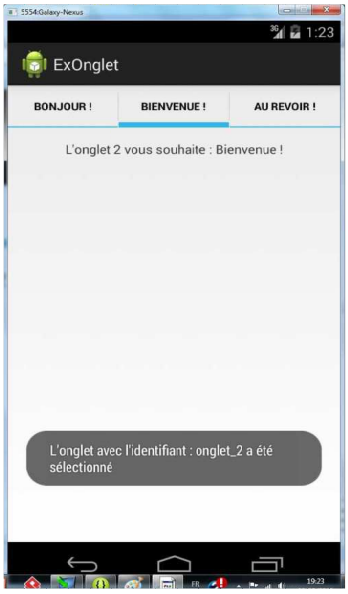
Le fichier XML aura donc la structure suivante :

```
<!-- TabHost contiendra la totalité des onglets -->
<TabHost
      xmlns:android="http://schemas.android.com/apk/res/android"
      android:id="@+id/tableOnglet"
      ..............                >

      <LinearLayout
          ...................       >

            <!-- TabWidget sert à afficher la ligne des boutons d'onglets -->
            <TabWidget
                    android:id="@android:id/tabs"
                    android:layout_width="match_parent"
                    android:layout_height="wrap_content">
            </TabWidget>

            <!-- Contenu de tous les  onglets -->
            <FrameLayout
                    android:id="@android:id/tabcontent"
                    android:layout_width="match_parent"
                    android:layout_height="match_parent">

                    <!-- Contenu de l'onglet numéro 1 -->
                    <LinearLayout
                    android:id="@+id/onglet1"
                            ..........................  />
                    </LinearLayout>

                    <!-- Contenu de l'onglet numéro 2 -->
                    <LinearLayout
                    android:id="@+id/onglet2"
                            ..............  >
                    </LinearLayout>

                    <!-- Contenu de l'onglet numéro 3 -->
                    <LinearLayout
                    android:id="@+id/onglet3"
                            ............              >
```

```
                    </LinearLayout>
                </FrameLayout>
            </LinearLayout>
</TabHost>
```

Exemple

Nous souhaitons développer une application dotée d'une seule activité comportant 3 onglets. Notons que lorsque l'utilisateur active le deuxième onglet, un message de type *Toast* doit s'afficher conformément à l'exemple ci-dessous.



Le fichier *XML* définissant la vue est le suivant :

```
<!-- TabHost contiendra la totalité des onglets -->
<TabHost
      xmlns:android="http://schemas.android.com/apk/res/android"
      android:id="@+id/tableOnglet"
      android:layout_width="match_parent"
      android:layout_height="match_parent">

      <LinearLayout
               android:orientation="vertical"
               android:layout_width="match_parent"
               android:layout_height="match_parent">

               <!-- TabWidget sert à afficher les onglets -->
               <TabWidget
                   android:id="@android:id/tabs"
                   android:layout_width="match_parent"
                   android:layout_height="wrap_content">
               </TabWidget>

               <!-- Contenu des onglets -->
               <FrameLayout
                        android:id="@android:id/tabcontent"
                        android:layout_width="match_parent"
                        android:layout_height="match_parent">

                        <!-- Contenu de l'onglet numéro 1 -->
                        <LinearLayout
                             android:id="@+id/onglet1"
                             android:orientation="vertical"
                             android:layout_width="match_parent"
                             android:layout_height="match_parent">
                             <TextView
                                   android:layout_width="wrap_content"
                                   android:layout_height="wrap_content"
                                   android:layout_marginTop="15dip"
```

```
                        android:layout_gravity="center"
                        android:text="@string/contenu_onglet1" />
                </LinearLayout>

                <!-- Contenu de l'onglet numéro 2 -->
                <LinearLayout
                      android:id="@+id/onglet2"
                      android:orientation="vertical"
                      android:layout_width="match_parent"
                      android:layout_height="match_parent">
                      <TextView
                            android:layout_width="wrap_content"
                            android:layout_height="wrap_content"
                            android:layout_marginTop="15dip"
                            android:layout_gravity="center"
                            android:text="@string/contenu_onglet2" />
                </LinearLayout>

                <!-- Contenu de l'onglet numéro 3 -->
                <LinearLayout
                     android:id="@+id/onglet3"
                     android:orientation="vertical"
                     android:layout_width="match_parent"
                     android:layout_height="match_parent">
                     <TextView
                           android:layout_width="wrap_content"
                           android:layout_height="wrap_content"
                           android:layout_marginTop="15dip"
                           android:layout_gravity="center"
                           android:text="@string/contenu_onglet3" />
                </LinearLayout>
            </FrameLayout>
        </LinearLayout>
</TabHost>
```

Dans le code Java, la démarche est la suivante :

1. récupérer l'instance de `TabHost` grâce à l'appel à la méthode ***findViewById***

2. appliquer la méthode ***setUp()*** sur cette instance (obligatoire) avant toute autre opération

3. il faut ensuite indiquer à l'instance de `TabHost` quelles sont les vues qui représentent le contenu des onglets et à quoi doit ressembler la ligne des boutons d'onglet. Pour ce faire, il faut initialiser un objet de type `TabSpec` pour chacun des onglets avec les valeurs adéquates :
   - ✓ un marqueur interne de l'onglet grâce à l'appel à ***newTabSpec***
   - ✓ le contenu de l'onglet grâce à l'appel à ***setContent*** : on donne en argument l'identifiant de l'onglet `android:id` présent dans la vue
   - ✓ le titre de l'onglet, et éventuellement une icône, grâce à la méthode ***setIndicator***

4. Cet objet doit ensuite être ajouté à l'instance de type `TabHost` (méthode ***addTab***)

5. On peut ensuite spécifier quel onglet s'affichera initialement (par défaut ce sera le premier)

```
public class MainActivity extends Activity {

    /**
     * Table d'onglets gérée par l'activité
     */
    private TabHost lesOnglets;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
```

```
        // on récupère la table d'onglets définie dans la vue
        lesOnglets = (TabHost) findViewById(R.id.tableOnglet);
        lesOnglets.setup();         // obligatoire ici

        // on ajoute les 3 onglets à l'instance de type TabHost

        /*
         *  Etape 1 : on décrit le premier onglet et on l'ajoute au TabHost
         *  Il faut spécifier :
         *     - son marqueur interne (appel à newTabSpec)
         *     - la chaîne de caractères qui sera affichée dans le bouton de
         *         l'onglet (appel à setIndicator)
         *     - l'identifiant la vue associée à l'onglet (appel à setContent)
         */
        TabSpec specification = lesOnglets.newTabSpec("onglet_1");
        specification.setIndicator(getResources().getString(R.string.label_onglet1));
        specification.setContent(R.id.onglet1);
        lesOnglets.addTab(specification);

        /*
         *  Etapes 2 et 3 : on décrit le deuxième et le troisième onglet
         *  et on les ajoute. Version de code plus concise
         */
        lesOnglets.addTab(lesOnglets.newTabSpec("onglet_2")
                                    .setIndicator(getResources().getString(R.string.label_onglet2)  )
                                    .setContent(R.id.onglet2));
        lesOnglets.addTab(lesOnglets.newTabSpec("onglet_3")
                                    .setIndicator(getResources().getString(R.string.label_onglet3))
                                    .setContent(R.id.onglet3));

        // on paramètre un écouteur pour écouter les changements d'onglet
        lesOnglets.setOnTabChangedListener(
             new TabHost.OnTabChangeListener() {

                 @Override
                 public void onTabChanged(String tabId) {

                     /*
                      * un message de type Toast sera affiché lors d'un
                      * changement d'onglet
                      */
                     Toast.makeText(MainActivity.this,
                                    "L'onglet avec l'identifiant : "+ tabId + " a été sélectionné",
                                    Toast.LENGTH_SHORT).show();
                  }
              });

        /*
         * on précise qu'au lancement de l'activité, c'est le premier onglet qui est
         * affiché. Remarque : l'appel ci-dessous est inutile puisque par défaut
         * c'est le premier onglet qui sera affiché
         */
        lesOnglets.setCurrentTab(0);
    }
}
```

Remarques

Tout comme il existe une classe `ListActivity`, il existe une classe `TabActivity` dont une activité basée sur les onglets peut hériter.

Dans le fichier *XML* définissant la vue, si un onglet contient un seul *widget*, il n'est pas nécessaire de définir un *layout* pour le contenu de cet onglet. Le *widget* viendra se placer juste en dessous de l'élément `FrameLayout`. Nous aurions pu utiliser cette possibilité dans l'exemple.

V) Les menus

Les activités Android peuvent gérer 2 types de menus :

- les **menus d'options**. Des terminaux Android disposent d'une touche dédiée à leur ouverture, ou alors une autre possibilité est de faire apparaître l'icône d'activation dans la barre d'action
- les **menus contextuels**. Ce type de menu s'ouvre lorsque l'utilisateur touche le *widget* doté de ce type de menu pendant un temps suffisamment long.

5.1) Les menus contextuels

Pour doter un *widget* d'un menu contextuel, il faut d'abord définir un fichier *XML* décrivant le menu et les options qu'il comporte. Bien sûr, toutes les indications données dans le fichier *XML* pourraient être fournies dans le code Java. Mais le code obtenu sera plus simple et plus lisible, si la définition du menu est faite dans le fichier XML.

D'une manière générale, ce fichier décrivant un menu aura la structure suivante :

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
      <item
          android:id="@+id/option1"
          android:title="@string/libelle_option1" />
      <item
          android:id="@+id/option2"
          android:title="@string/libelle_option2" />
      <item
          android:id="@+id/option3"
          android:title="@string/libelle_option3" />
</menu>
```

Structure d'un fichier *XML* décrivant un menu

D'autres propriétés pourraient être précisées, comme par exemple `android:icon` pour associer une icône à une option ou `android:visible` pour rendre visible ou non une option selon les besoins. La propriété définissant un identifiant est obligatoire pour pouvoir ensuite retrouver l'option que l'utilisateur a choisie dans le menu.

Il est possible de définir des sous-menus dans un menu. Dans ce cas, un élément *menu* prendra la place d'un *item*.

Dans la classe Java associée à l'activité, il faudra, dans la méthode *onCreate*, indiquer quels sont les *widgets* qui vont disposer d'un menu. Cette indication s'effectue en invoquant la méthode :

void registerForContextMenu(View view)

On donne en paramètre de cette méthode la vue (ou *widget*) à laquelle on souhaite associer un menu. Cette méthode est définie dans la classe *Activity*.

Dans la classe Java associée à l'activité, il faudra redéfinir impérativement 2 méthodes : ***onCreateContextMenu*** et ***onContextItemSelected.***

→ La première des deux méthodes a le profil suivant :

void onCreateContextMenu(ContexMenu menu, View v, ContextMenu.ContextMenuInfo menuInfo)

Cette méthode est appelée automatiquement chaque fois que l'utilisateur provoquera l'activation de l'un des menus contextuels présents dans l'activité. Les paramètres de la méthode ont la signification suivante :

- *menu* est le menu activé par l'utilisateur
- *v* est la vue (ou *widget*) à laquelle le menu est associé
- *menuInfo* contient des informations supplémentaires sur la vue à laquelle le menu est associé. Ces informations dépendent du type de la vue.

Si on n'a pas défini un fichier *XML* décrivant le menu, c'est dans cette méthode que l'on place les instructions Java permettant de positionner les options du menu. On les ajouterait au paramètre *menu*.

Si par contre, on a défini un fichier *XML* décrivant le menu, il suffira d'écrire dans la méthode l'instruction permettant de désérialiser le code *XML* décrivant le menu, et de l'associer au menu activé.

Les menus contextuels sont supprimés après utilisation. Il faut les reconstruire chaque fois que l'utilisateur les active. A chaque reconstruction, la méthode ***onCreateContextMenu*** est invoquée automatiquement.

→ La deuxième méthode à redéfinir a le profil suivant :

boolean onContextItemSelected(MenuItem item)

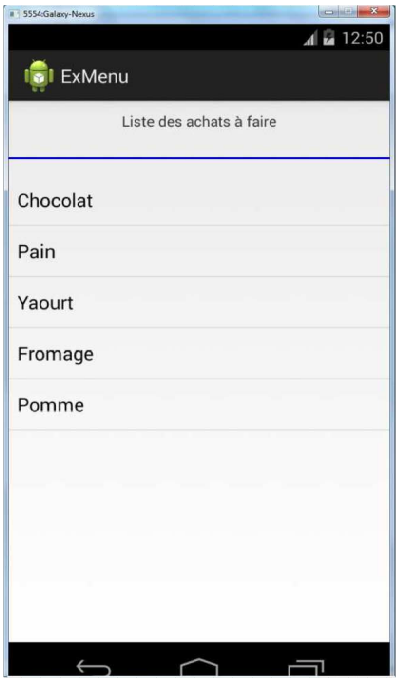
Cette méthode est appelée automatiquement lorsque l'utilisateur sélectionnera une option d'un menu. Le paramètre de la méthode fournit cette option. Si l'activité comporte plusieurs menus contextuels, il faut donc que toutes les options aient des identifiants différents.

Il est possible d'appliquer à *item* la méthode ***getMenuInfo().*** Elle renverra une instance de ***ContextMenu.ContextMenuInfo*** contenant des informations sur la vue à laquelle le menu est associé.

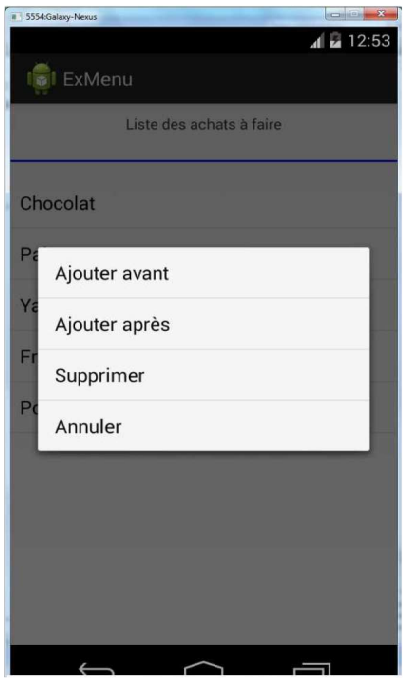
Exemple

On souhaite gérer une liste d'achats à faire. Au lancement de l'activité, une liste prédéfinie sera affichée. Lorsque l'utilisateur fera un clic long sur l'un des achats de la liste, un menu contextuel apparaîtra et proposera des options conformément à l'exemple suivant.

Ecran 1

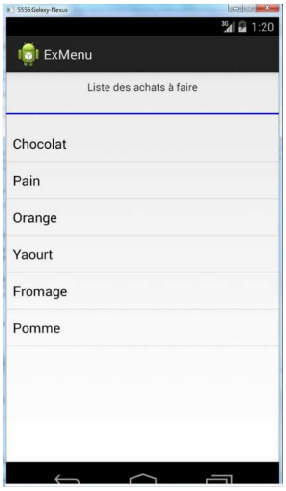


Ecran 2



Ecran 3

Les chaînes de caractères sont définies de la manière suivante :




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

<string name="app_name">ExMenu</string>
<string name="message_introduction">Liste des achats à faire</string>

<!--  libellés des options du menu -->
<string name="texte_ajouter_avant">Ajouter avant</string>
<string name="texte_ajouter_apres">Ajouter après</string>
<string name="texte_supprimer">Supprimer</string>
<string name="texte_annuler">Annuler</string>

<!--  messages affichés dans la boîte permettant de saisir
un nouvel article -->
<string name="titre_boite">"Ajouter un article à la liste"</string>
<string name="message_saisie">"Nom de l'article à ajouter :"</string>
<string name="message_aide_saisie">"Tapez le nom de l'article"</string>
<string name="bouton_positif">"Valider"</string>
<string name="bouton_negatif">"Annuler"</string>

</resources>
```

Fichier *strings.xml*

Notre application comportera 2 fichiers *layout*. Le premier définit le *layout* associé à l'activité elle-même. Le deuxième définit la mise en forme de la boîte de dialogue permettant de saisir l'article à ajouter à la liste.

```xml
<!--  Vue associée à l'activité principale : affiche un message de bienvenue,
      une barre horizontale et une liste (d'achats à effectuer) -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
```

```
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="${relativePackage}.${activityClass}"
    android:orientation="vertical" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_marginTop="10dip"
        android:layout_marginBottom="10dip"
        android:text="@string/message_introduction" />

    <View
         android:layout_width="fill_parent"
        android:layout_height="2dip"
        android:layout_marginTop="15dip"
        android:layout_marginBottom="15dip"
        android:background="#0000FF" />

    <ListView
        android:id="@android:id/list"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:drawSelectorOnTop="false" />

</LinearLayout>
```

Fichier *activite_liste_menu.xml* (vue associée à l'activité principale)

```
<!--  Vue associée à la boîte de dialogue permettant d'effectuer la saisie d'un article.
      Cette vue comporte :
          une étiquette invitant l'utilisateur à effectuer la saisie
           une zone de saisie -->

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="10dip"
        android:layout_marginTop="10dip"
        android:layout_marginBottom="10dip"
        android:text="@string/message_saisie" />

    <EditText
        android:id="@+id/saisieArticle"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/message_aide_saisie" />

</LinearLayout>
```

Fichier *saisie_article.xml* (vue associée à la boîte permettant de saisie)

Le fichier décrivant le menu est un fichier XML nommé ***menusurachat.xml***. Il se trouve dans le dossier ***res***, et le sous-dossier ***menu***.

```
<?XML version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
      <item
          android:id="@+id/ajout_avant"
          android:title="@string/texte_ajouter_avant" />
      <item
          android:id="@+id/ajout_apres"
          android:title="@string/texte_ajouter_apres" />
      <item
          android:id="@+id/suppression"
          android:title="@string/texte_supprimer" />
      <item
          android:id="@+id/annuler"
          android:title="@string/texte_annuler" />
</menu>
```

Fichier *menusurachat.xml* (décrit le menu contextuel et ses options)

Le code Java de la classe décrivant l'activité sera le suivant.

```
/*
 * Affiche une liste d'achats que l'utilisateur pourra modifier
 * ActiviteListeMenu.java                                                    02/15
 */
package com.example.exmenu;

import java.util.ArrayList;
import java.util.Arrays;
.....

/**
 * L'activité affiche une liste d'achats préétablie.
 * L'utilisateur a ensuite la possibilité de rajouter des articles à acheter
 * et aussi d'en supprimer.
 * Ces modifications de la liste s'effectuent via un menu contextuel.
 * Version intermédiaire : les articles de la liste ne sont pas enregistrés
 * @author LP MMS
 * @version 1.0
 */
public class ActiviteListeMenu extends ListActivity {


      /** Elements qui seront présents dans la liste de base */
      private static final String[] LISTE_BASE = {"Chocolat", "Pain",
                                                  "Yaourt", "Fromage", "Pomme"};

      /** Liste contenant les achats à afficher */
      private ArrayList<String> listeAchat;

      /** Adaptateur permettant de gérer la liste des achats */
      private ArrayAdapter<String> adaptateur;



      @Override
      protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.activite_liste_menu);
        listeAchat = new ArrayList<>(Arrays.asList(LISTE_BASE));

        // On crée un adpateur pour rassembler les données à afficher
        adaptateur = new ArrayAdapter<String>(this,
                                              android.R.layout.simple_list_item_1,
                                              listeAchat);
        setListAdapter(adaptateur);

        // on précise qu'un menu est associé à la liste qui correspond à l'activité
        registerForContextMenu(getListView());
    }

    /**
     * Méthode invoquée automatiquement lorsque l'utilisateur active un menu contextuel
     */
    @Override
    public void onCreateContextMenu(ContextMenu menu, View v,
                                    ContextMenu.ContextMenuInfo menuInfo) {

        /*
         *  on désérialise le fichier XML décriant le menu et on l'associe
         *  au menu argument (celui qui a été activé)
         */
        new MenuInflater(this).inflate(R.menu.menusurachat, menu);
            }


    /**
     * Méthode invoquée automatiquement lorsque l'utilisateur choisira une option
     * dans un menu contextuel
     */
    @Override
    public boolean onContextItemSelected(MenuItem item) {

        /*
         *  on accède à des informations supplémentaires sur la vue associée
         *  au menu activé. L'information qui nous intéresse est la position
         *  de l'élément de la liste sur lequel l'utilisateur a cliqué pour
         *  activer le menu.
         */
        AdapterView.AdapterContextMenuInfo information =
                    (AdapterView.AdapterContextMenuInfo) item.getMenuInfo();

        // selon l'option sélectionnée dans le menu, on réalise le traitement adéquat
        switch(item.getItemId()) {
        case R.id.ajout_avant :   // ajout d'un article avant l'article courant
               saisirArticleAAjouter(information.position);
               break;
        case R.id.ajout_apres:   // ajout d'un article après l'article courant
               saisirArticleAAjouter(information.position + 1);
               break;
        case R.id.suppression :   // suppression de l'article courant

               // on supprime de l'adaptateur l'article courant
               adaptateur.remove(listeAchat.get(information.position));
               break;
        case R.id.annuler :           // retour à la liste principale
               break;

        }
        return (super.onContextItemSelected(item));
    }


    /**
     * Cette méthode affiche une boîte de dialogue permettant à l'utilsiateur de
```

```
         * saisir un nouvel article. Cet article est ensuite ajouté à la liste gérée
         * par l'activité
         * @param positionAjout  un entier égal à une position dans la liste.
         *                          L'ajout de l'article saisi se fera sur cette position
         */
        private void saisirArticleAAjouter(final int positionAjout) {

                // on désérialise le layout qui est associé à la boîte de saisie
                final View boiteAjoutArticle =
                                getLayoutInflater().inflate(R.layout.saisie_article, null);

                /*
                 * Création d'une boîte de dialogue :
                 *   - on positionne un titre
                 *   - on lui associe une vue (le layout précédemment désérialisé)
                 *   - on associe un libellé et un comportement au bouton positif :
                 *          on récupère le texte tapé par l'utilisateur dans l'EditText
                 *          on modifie la liste des achats, pour lui ajouter l'article saisi
                 *             à la position du clic, ou juste après la position du clic
                 *          on informe l'adaptateur qu'il y a une modification de liste qu'il
                 *             doit prendre en compte
                 *   - on associe un libellé et un comportement (null) au bouton négatif
                 *   - on rend la boîte visible
                 */
                new AlertDialog.Builder(this)
                    .setTitle(getResources().getString(R.string.titre_boite))
                    .setView(boiteAjoutArticle)
                    .setPositiveButton(getResources().getString(R.string.bouton_positif),
                                       new DialogInterface.OnClickListener() {
                       public void onClick(DialogInterface dialog,
                                           int leBouton) {
                               EditText article = (EditText)
                                      boiteAjoutArticle.findViewById(R.id.saisieArticle);
                               String articleSaisi = article.getText().toString();
                               listeAchat.add(positionAjout,articleSaisi);
                               adaptateur.notifyDataSetChanged();
                       }
                    })
                    .setNegativeButton(getResources().getString(R.string.bouton_negatif), null)
                    .show();
        }
}
```

Fichier *ActiviteListeMenu.java* (activité principale)

5.2) Les menus d'options

Ce type de menu peut s'ouvrir soit si on appuie sur le bouton *Menu* du terminal (bouton physique qui n'est pas situé sur l'écran du terminal, mais en dessous de celui-ci), soit si on l'active en appuyant sur l'icône "3 points" de la **barre d'action** située en haut de l'écran du terminal. Cette barre d'action existe depuis Android 3.0. Elle se place automatiquement en haut de l'écran de l'application, sauf indication contraire de la part du programmeur. On voit apparaître dans cette barre : l'icône de l'application, éventuellement le nom de l'application, des possibilités pour naviguer entre les vues de l'application, des boutons d'actions et l'icône d'ouverture du menu d'options (« 3 points »).

Notons qu'un seul menu d'options peut exister pour une activité donnée, contrairement aux menus contextuels.

Pour doter l'application d'un tel menu, il faut implémenter 2 méthodes de la classe *Activity* : ***onCreateOptionsMenu*** et ***onOptionsItemSelected.***

→ La première des deux méthodes a le profil suivant :

boolean onCreateOptionsMenu(Menu menu)

Cette méthode sera appelée automatiquement lors de la première activation du menu d'options. C'est dans cette méthode que l'on peut appliquer différentes méthodes à l'instance de type *Menu* pour le décrire (placer les options, les icônes éventuellement, des raccourcis ...), ou alors effectuer un appel à la méthode ***inflate*** pour désérialiser le menu décrit dans un fichier *XML* (voir exemple).

→ La deuxième des deux méthodes a le profil suivant :

boolean onOptionsItemSelected(MenuItem item)

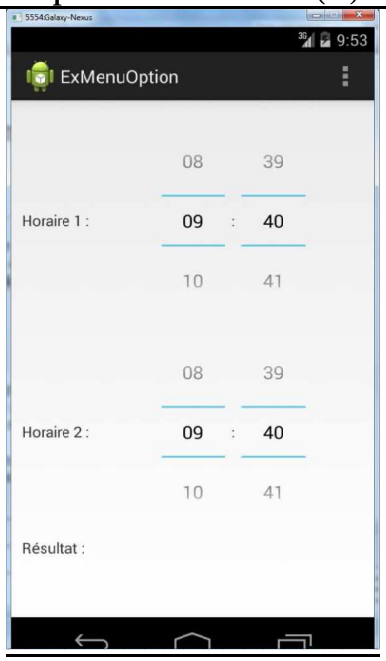
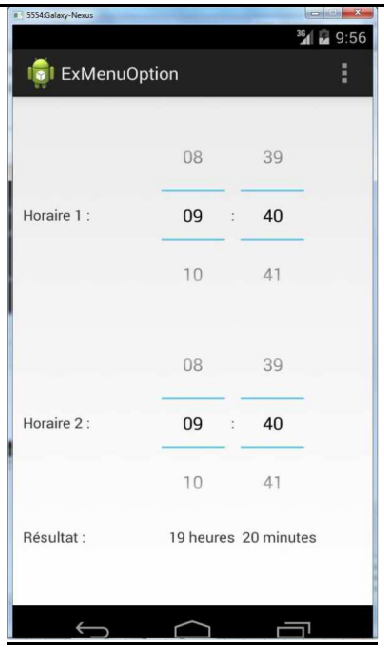
Cette méthode sera appelée automatiquement chaque fois que l'utilisateur cliquera sur un item du menu d'options. Il faut écrire dans cette méthode le traitement à réaliser selon le choix de l'utilisateur. Le paramètre *MenuItem* décrit l'item sélectionné par l'utilisateur. Comme pour les menus contextuels, la méthode ***getItemId()*** appliqué à l'instance *MenuItem* renvoie l'identifiant de l'item sélectionné.

Si on doit modifier le menu au cours de l'activité (désactiver ou ajouter des options par exemple), on peut le faire soit via l'instance de type *Menu* récupérée lors de l'appel à *onCreateOptionsMenu*, soit en redéfinissant la méthode ***boolean onPrepareOptionsMenu(Menu menu).*** Celle-ci est invoquée automatiquement avant chaque affichage du menu.

Exemple

L'application permettra à l'utilisateur de saisir 2 durées exprimées en heures et minutes. Celui-ci devra ensuite, à travers un menu d'options, choisir l'opération qu'il souhaite réaliser :

- ✓ calculer l'écart entre les deux durées exprimé en minutes
- ✓ calculer l'écart en heures et minutes
- ✓ ajouter les deux durées pour obtenir une durée en minutes
- ✓ ajouter pour obtenir une durée en heures et minutes
- ✓ ou bien annuler

Au lancement de l'application, remarquer dans l' "***action bar***", l'option située à droite (...)	L'utilisateur clique pour ouvrir le menu d'options, et il sélectionne l'option *Ajouter*	Le menu se referme, et l'application affiche le résultat
		

Pour obtenir cette application, il faut définir le fichier *strings.xml* de la manière suivante :

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

    <string name="app_name">ExMenuOption</string>
    <string name="libelle_horaire1">"Horaire 1 : "</string>
    <string name="libelle_horaire2">"Horaire 2 : "</string>
    <string name="libelle_resultat_fixe">"Résultat : "</string>

    <!--  libellés des options du menu d'option -->
    <string name="texte_ecart_minute">Ecart en minutes</string>
    <string name="texte_ecart_heure">Ecart en heures</string>
    <string name="texte_ajouter">Ajouter</string>
    <string name="texte_ajouter_minute">Ajouter en minutes</string>
    <string name="texte_annuler">Annuler</string>

</resources>
```

Fichier *strings.xml*

On définit également un fichier *XML* pour décrire le menu d'options :

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

      <item
          android:id="@+id/ecart_heure"
          android:title="@string/texte_ecart_heure" />
      <item
          android:id="@+id/ecart_minute"
          android:title="@string/texte_ecart_minute" />
      <item
          android:id="@+id/ajouter_heure"
          android:title="@string/texte_ajouter" />
      <item
```

```
            android:id="@+id/ajouter_minute"
            android:title="@string/texte_ajouter_minute" />
        <item
            android:id="@+id/annuler"
            android:title="@string/texte_annuler" />
</menu>
```

Fichier *menuhoraire.xml*

Le fichier *XML* décrivant la vue de l'activité est écrit comme suit :

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:stretchColumns="1"
    tools:context="${relativePackage}.${activityClass}" >

    <!--  Première ligne : Saisie du premier horaire -->
    <TableRow
        android:layout_marginTop="15dip" >
        <TextView
            android:layout_marginLeft="10dip"
            android:layout_gravity="center_vertical"
            android:textSize="15sp"
            android:text="@string/libelle_horaire1"/>
        <TimePicker
             android:id="@+id/horaire1"
       android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:layout_gravity="center" />
    </TableRow>

    <!--  Deuxième ligne : Saisie du deuxième horaire -->
    <TableRow>
        <TextView
            android:layout_marginLeft="10dip"
            android:layout_gravity="center_vertical"
            android:textSize="15sp"
            android:text="@string/libelle_horaire2"/>
        <TimePicker
            android:id="@+id/horaire2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center" />
    </TableRow>

    <!--  Troisième ligne : Affichage du résultat -->
    <TableRow>
        <TextView
            android:layout_marginLeft="10dip"
            android:textSize="15sp"
            android:text="@string/libelle_resultat_fixe"/>
        <TextView
            android:id="@+id/zone_resultat"
            android:layout_marginLeft="10dip"
            android:textSize="15sp"
            android:layout_gravity="center"
            android:layout_column="1"/>
     </TableRow>

</TableLayout>
```

Fichier *gestion_horaire.xml*

La classe Java décrivant l'activité est la suivante :

```
/*
 * Cette activité permet à l'utilisateur d'effectuer des opérations sur 2 horaires
 * GestionHoraire.java
                02/15
 */
package com.example.exmenuoption;

import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.widget.TextView;
import android.widget.TimePicker;

/**
 * Cette activité permet à l'utilisateur de saisir 2 horaires, grâce au widget
 * TimePicker.
 * L'utilisateur choisit ensuite l'opération qu'il souhaite réaliser via un
 * menu d'options qui apparaîtra dans la barre d'action de l'activité.
 * Les opérations possibles sont :
 *    calculer l'écart entre les 2 horaires, soit en minutes, soit en heures et minutes
 *    calculer la somme des 2 horaires, soit en minutes, soit en heures et minutes
 * @author LP MMS
 * @version 1.0
 */
public class GestionHoraire extends Activity {

    /** Widget pour saisir le premier horaire */
    private TimePicker horaire1;

    /** Widget pour saisir le deuxième horaire */
    private TimePicker horaire2;

    /** Widget pour afficher le résultat de l'opération demandée */
    private TextView resultat;


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.gestion_horaire);

        // on récupère un accès sur les widgets de la vue
        horaire1 = (TimePicker) findViewById(R.id.horaire1);
        horaire2 = (TimePicker) findViewById(R.id.horaire2);
        resultat = (TextView) findViewById(R.id.zone_resultat);

        // les horaires seront saisis avec un nombre d'heures compris entre 0 et 23
        horaire1.setIs24HourView(true);
        horaire2.setIs24HourView(true);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        new MenuInflater(this).inflate(R.menu.menuhoraire, menu);
        return super.onCreateOptionsMenu(menu);
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        int heure1 = 0, heure2 = 0,               // valeurs saisies et récupérés
            minute1 = 0 , minute2 = 0,
            resultatEnMinute;                     // résultat calculé
```

```
        // si l'utilisteur n'a pas choisi annuler : récupérer les horaires saisis
        if (item.getItemId() != R.id.annuler) {
            heure1 = horaire1.getCurrentHour();
            minute1 = horaire1.getCurrentMinute();
            heure2 = horaire2.getCurrentHour();
            minute2 = horaire2.getCurrentMinute();
        }

        // réaliser l'opération souhaitée par l'utilisateur
        switch (item.getItemId()) {
        case R.id.ecart_minute :           // calcul de l'écart des horaires en minutes
            resultatEnMinute = OutilDureeHM.ecart(heure1, minute1, heure2, minute2);
            resultat.setText(resultatEnMinute + " minute(s)");
            break;
        case R.id.ecart_heure :            // calcul de l'écart des horaires en heures
            resultatEnMinute = OutilDureeHM.ecart(heure1, minute1, heure2, minute2);
            resultat.setText(OutilDureeHM.formater(resultatEnMinute));
            break;
        case R.id.ajouter_heure :          // calcul de la somme des horaires
            resultatEnMinute = OutilDureeHM.ajouter(heure1, minute1, heure2, minute2);
            resultat.setText(OutilDureeHM.formater(resultatEnMinute));
            break;
        case R.id.ajouter_minute :         // calcul de la somme des horaires
            resultatEnMinute = OutilDureeHM.ajouter(heure1, minute1, heure2, minute2);
            resultat.setText(resultatEnMinute + " minute(s)");
            break;
        case R.id.annuler :                // annuler
            break;
        }

        return super.onOptionsItemSelected(item);
    }
}
```

Fichier *GestionHoraire.java*

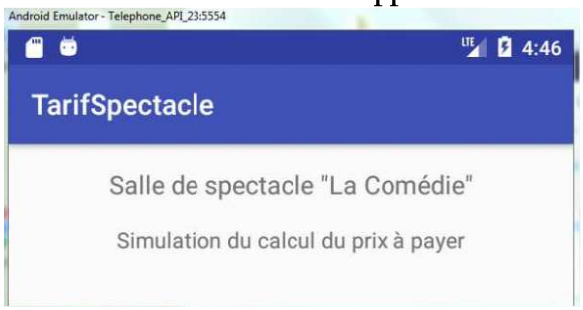
6) La barre d'action et la ToolBar

6.1) La barre d'action – Android 3.0

La notion de barre d'action existe depuis Android 3.0. Cette barre d'action (nommée aussi *action bar* ou *app bar*) se place automatiquement en haut de l'écran de l'application, sauf indication contraire de la part du programmeur. On voit apparaître dans cette barre :

- l'icône de l'application. Mais cet affichage n'est plus conseillé dans les versions récentes
- le nom de l'application,
- des possibilités pour naviguer entre les vues de l'application,
- des boutons d'actions
- et l'icône d'ouverture du menu d'options (« 3 points »).

Exemple avec une barre contentant seulement le nom de l'application :



Les applications créées automatiquement par Android Studio sont dotées d'une barre d'action. Tout est mis en œuvre pour que la barre d'action soit présente même si l'application est installée sur un terminal doté d'une version d'Android antérieure à la 3.0, celle qui a intégrée la barre d'action. La rétro-compatibilité est assurée grâce à la bibliothèque de support AndroidSupport v7. En résumé, pour que la barre d'action soit affichée quelle que soit la version d'Android, il faut que les 3 éléments suivants soient mis en œuvre :

1) Dans le fichier *build.gradle*, une dépendance doit être présente :

```
dependencies {
        implementation 'com.android.support:appcompat-v7:27.1.1'
```

Il s'agit d'une dépendance envers la bibliothèque de support AndroidSupport v7. Celle-ci fournit des classes pour rendre l'application développée compatible avec les versions antérieures d'Android (rétro-compatibilité), plus précisément les versions antérieures à l'API 27.

2) Le style de l'application doit prévoir une barre d'action. Dans le fichier *styles.xml*, on trouve la ligne:

```
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
```

3) Au niveau du code Java, la classe activité ne doit pas hériter de *Activity* mais de *AppCompatActivity*/ Cette classe est présente dans la bibliothèque support AndroidSupport v7. C'est un équivalent de la classe *Activity* qui assure en plus la rétro-compatibilité :

```
public class MainActivity extends AppCompatActivity {
```

6.2) *ToolBar* à partir d'Android 5.0

Android 5.0 a apporté une nouveauté au niveau de la barre d'actions avec la notion de ***ToolBar***. Ce composant donne plus de souplesse et de possibilité de personnalisation que *l'ActionBar*. On pourra ajouter dans celle-ci, par exemple, des images ou du texte sous la forme des *widgets* classiques *ImageView* ou *TextView*.

La ***ToolBar*** est un composant *widget* au même titre que les autres *widgets*, *TextView*, *Button* … La ***Toolbar*** se trouve dans la bibliothèque de compatibilité v7 qu'il faudra donc inclure dans notre projet, comme indiqué précédemment.

6.2.1) Comment faire apparaître une *ToolBar* ?

1) Il faut d'abord intégrer un widget ***Toolbar*** dans le fichier *layout* décrivant la vue de l'activité. Le mieux est d'utiliser le *widget* provenant de la bibliothèque de compatibilité selon l'exemple suivant (mais il existe aussi un *widget* ayant pour nom `android.widget.Toolbar`) :

```
<android.support.v7.widget.Toolbar
        android:id="@+id/ma_tool_bar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="?attr/colorPrimary"
        android:elevation="4dp" />
```

L'attribut *elevation* fait partie des recommandations de *Material Design* (= ensemble des bonnes pratiques qu'il est recommandé de suivre en matière de design pour obtenir des interfaces adaptées à tous les types d'appareils). La valeur de l'attribut détermine l'importance de l'ombre portée qui sera ajoutée lors de l'affichage de la *Toolbar*. Google préconise une élévation de 4dp pour le composant *Toolbar*.

De même, Google préconise une hauteur précise pour le composant *Toolbar*. Cette hauteur est définie via la constante *actionBarSize*, constante prédéfinie par la plateforme Android.

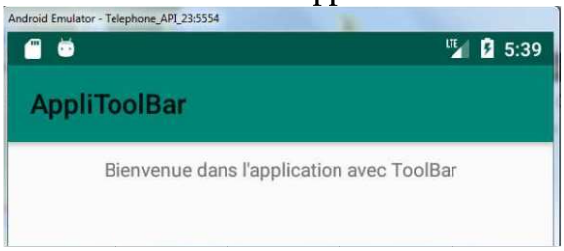
2) Le programmeur doit veiller à bien choisir le thème de l'application. En effet, il y a un risque que l'application affiche à la fois la barre d'action et la *Toolbar*. Le programmeur doit donc soit choisir un thème dépourvu de barre d'action, comme par exemple `AppCompat.NoActionBar`, soit préciser dans le thème que ni la barre d'action, ni le titre de celle-ci (sous-entendu dans la barre d'action) ne doivent s'afficher :

```
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
      <!-- Customize your theme here. -->
      <item name="android:windowActionBar">false</item>
      <item name="windowNoTitle">true</item>
```

3) Dans le code Java de l'activité, il faut préciser quel élément du fichier *layout* va jouer le rôle de *Toolbar*. Tout d'abord, la classe doit hériter de *AppCompatActivity*. Ensuite on récupère un accès au *widget Toolbar* défini dans le fichier *layout* et on associe cette *Toolbar* à l'activité en invoquant la méthode ***setSupportActionBar***

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Toolbar maBarreOutil = findViewById(R.id.ma_tool_bar);
        setSupportActionBar(maBarreOutil);
    }
```

Par défaut, la *Toolbar* affiche seulement le nom de l'application.



6.2.2) Comment personnaliser une *ToolBar* ?

Les boutons d'action sont en fait des raccourcis, sous forme d'icône, vers les fonctionnalités les plus importantes de l'application. Ces mêmes fonctionnalités peuvent être accessibles via le menu d'options. Si l'on souhaite faire apparaître des boutons d'action, il faut agir sur le fichier XML contenant la description du menu d'options.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto">

       <item
           android:id="@+id/premiere_option"
           android:title="@string/texte_premiere_option"
           android:icon="@android:drawable/ic_menu_manage"
           android:orderInCategory="1"
           app:showAsAction="ifRoom"     />

       <item
           android:id="@+id/deuxieme_option"
           android:title="@string/texte_ deuxieme _option"
           android:icon="@android:drawable/ic_menu_share"
           android:orderInCategory="2"
           app:showAsAction="ifRoom"    />

       <item
           android:id="@+id/troisieme_option"
           android:title="@string/texte_ troisieme _option"
           android:icon="@android:drawable/ic_menu_search"
           android:orderInCategory="3"
           app:showAsAction="ifRoom"  />
</menu>
```

En plus d'un identifiant et d'un libellé, nous avons associé à chaque option du menu : une icône, un numéro d'ordre et la description du comportement de cette option au sein de la barre d'action.

L'icône peut être une icône créée par le programmeur et placée, dans ce cas dans le dossier des ressources. Mais elle peut être aussi une icône prédéfinie par Android, ce qui est le cas dans l'exemple. On reconnaît cette particularité car le nom de l'icône débute par *"@android:drawable.*

Les 3 icônes référencées dans l'exemple sont respectivement :

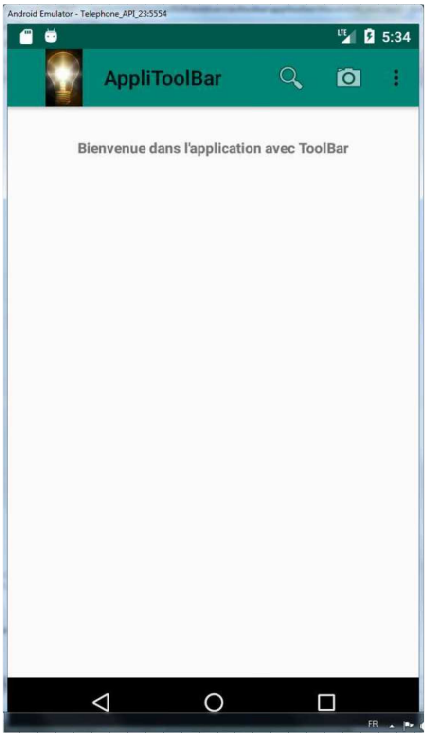
  

Remarque : le site Android Developer fournit la liste des icônes et/ou images prédéfinies disponibles selon les versions d'Android (on peut rechercher « Material icon »).

L'attribut `orderInCategory` permet de spécifier dans quel ordre les actions doivent être affichées. Cet ordre est lié aussi à l'attribut `showAsAction` . Nous avons donné à celui-ci la valeur *`ifRoom`* qui signifie que l'icône ne sera affichée dans la barre d'action que si la place est suffisante. D'autres valeurs sont possibles :

- *`withText`* : pour inclure le texte du libellé de l'action
- *`never`* : pour ne jamais placer l'option dans la barre d'actions
- *`always`* : pour que l'option soit toujours placée dans la barre d'actions (déconseillé, car si la place est insuffisante, les icônes vont se superposer)
- *`collapseActionView`* : si l'utilisateur active cette option, le nom de l'option apparaîtra dans la barre d'action et sera précédée par une flèche permettant de revenir à l'affichage normal :

6.2.3) Exemple



On souhaite que notre application affiche dans la ***Toolbar*** :

- une image
- un titre
- obligatoirement un bouton d'action (la loupe) pour effectuer une recherche
- éventuellement, s'il y a suffisamment de place, un bouton d'action pour activer l'appareil photo
- les 3 points permettant d'activer le menu d'option associé à l'application. On trouvera dans ce menu 3 options et éventuellement 4 si le bouton d'action de l'appareil photo n'a pas été affiché dans la barre, faute de place.

Lorsque l'utilisateur activera une option du menu, directement via un bouton d'action ou indirectement en passant par le menu d'option, un message de type *Toast* sera affiché.

On définit un fichier ***menu.xml*** comme indiqué ci-dessous :

```
<!-- Menu d'options de l'activité principale -->
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".MainActivity"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <!-- option : rechercher (icône loupe)
```

```
        l'icône représentant une loupe est définie parmi les icônes Android
           sous l'appellation ic_menu_search
        on précise avec l'attribut orderInCategory une priorité pour l'affichage
           de cette icône, sachant qu'elle ne s'affichera que s'il a assez de
           place (propriété showAsAction avec la valeur ifRoom)
        on précise également que lorsque l'option est activée une vue
        doit s'afficher. La vue précise est définie par la propriété
        actionViewClass -->
    <item
        android:id="@+id/premiere_option"
        android:title="@string/texte_premiere_option"
        android:icon="@android:drawable/ic_menu_search"
        android:orderInCategory="1"
        app:showAsAction="ifRoom|collapseActionView"
        app:actionViewClass="android.support.v7.widget.SearchView"/>

    <!-- option : camera (icône appareil photo)
        l'icône représentant un appareil photo est définie parmi
           les icônes Android sous l'appellation ic_menu_camera
        on précise avec l'attribut orderInCategory une priorité pour l'affichage
           de cette icône, sachant qu'elle ne s'affichera que s'il a assez de
           place (propriété showAsAction avec la valeur ifRoom).
           Son affichage est jugé moins prioritaire que celui de l'icône pour
           rechercher  -->
    <item
        android:id="@+id/deuxieme_option"
        android:icon="@android:drawable/ic_menu_camera"
        android:orderInCategory="2"
        android:title="@string/texte_deuxieme_option"
        app:showAsAction="ifRoom" />

    <!-- option : outil -->
    <item
        android:id="@+id/troisieme_option"
        android:title="@string/texte_troisieme_option"
        android:icon="@android:drawable/ic_menu_manage"
        app:showAsAction="never"  />

    <!-- option : partager -->
    <item
        android:id="@+id/quatrieme_option"
        android:title="@string/texte_quatrieme_option"
        android:icon="@android:drawable/ic_menu_share"
        app:showAsAction="never"  />

    <!-- option : supprimer -->
    <item
        android:id="@+id/cinquieme_option"
        android:title="@string/texte_cinquieme_option"
        android:icon="@android:drawable/ic_menu_delete"
        app:showAsAction="never"  />
</menu>
```

Dans le fichier ci-dessus, notez la présence du préfixe *app :* au lieu de *android :*. Ce préfixe désigne un espace de nom différent de celui utilisé habituellement (voir sa définition dans la balise *menu*). Cet espace de nom est lié à l'utilisation de la bibliothèque de compatibilité.

Notons également que la propriété *showAsAction* a la valeur *never* pour les 3 dernières options du menu. Donc ces 3 options n'apparaîtront pas directement dans la barre d'outil, mais elles seront visibles si on active le menu via les 3 points.

La vue de l'activité principale est définie comme suit :

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Vue de l'activité qui affiche une barre d'outil -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
```

```
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <!-- on utilise la barre d'outil de la bibliothèque de compatibilité
         pour assurer la rétrocompatibilité avec les anciennes versions
         d'Android -->
    <android.support.v7.widget.Toolbar
        android:id="@+id/ma_tool_bar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="?attr/colorPrimary"
        android:elevation="4dp"
        app:theme="@style/ThemeOverlay.AppCompat.ActionBar"
        app:popupTheme="@style/ThemeOverlay.AppCompat.Light">
    </android.support.v7.widget.Toolbar>

    <TextView
        android:id="@+id/texte_principal"
        android:layout_margin="30dp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:textStyle="bold"
        android:layout_gravity="center"
        android:text="@string/message_bienvenue"/>
</LinearLayout>
```

On souhaite rendre fonctionnel (au moins en partie) le bouton de recherche.

Lorsque l'utilisateur cliquera sur le bouton de recherche (la loupe), on souhaite voir apparaître dans la ToolBar une zone permettant d'effectuer la saisie du mot à chercher

Supposons que l'utilisateur tape le mot « java », puis clique sur la loupe du clavier virtuel pour lancer la recherche. On constate que l'activité a bien récupéré le mot recherché (puisqu'il est affiché)

Le code Java à écrire est le suivant :

```
/*
 * Classe activité pour illustrer la barre d'outil
 * ficheir MainActivity.java                                    01/19
 */
package tp2018.testdivers.testtoolbar.applitoolbar;

import . . .
/**
 * Cette activité comporte une barre d'outil.
 * Dans celle-ci, on trouve :
 *    - une image logo
 *    - le titre de l'application
 *    - une icone permettant de lancer une recherche (loupe)
 *    - une icone permettant d'utiliser la caméra
 *    - les trois points du menu d'options
 */
public class MainActivity extends AppCompatActivity  {

    /**
     * Message affiché par l'activité
     */
    private TextView texte;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        texte = findViewById(R.id.texte_principal);

        /*
         * On récupère un accès à la Toolbar définie dans le fichier layout.
         * On associe cette Toolbar à l'activité courante.
         * On place un logo sur la Toolbar (par défaut à gauche)
         */
        Toolbar maBarreOutil = findViewById(R.id.ma_tool_bar);
        setSupportActionBar(maBarreOutil);
        maBarreOutil.setLogo(R.drawable.iconeampoule);
    }


    /**
     * Méthode invoquée à la première activation du menu d'options
     * @param menuActivite   menu d'option activé
     * @return un booléen égal à vrai si le menu a pu être créé
     */
    @Override
    public boolean onCreateOptionsMenu(Menu menuActivite) {

        // on désérialise la vue associée au menu
        getMenuInflater().inflate(R.menu.menu, menuActivite);

        // on récupère un accès à l'option permettant de lancer une recherche
        MenuItem itemRecherche = menuActivite.findItem(R.id.premiere_option);

        /*
         * on récupère la vue affichée lors du lancement d'une recherche
         * (si l'utilisateur clique sur la loupe)
         * on modifie cette vue pour afficher une phrase d'indication dans la zone
         * de saisie du mot à chercher
         * on associe un écouteur à la zone de saisie
         */
        SearchView vuePourRecherche = (SearchView) itemRecherche.getActionView();
        vuePourRecherche.setQueryHint(getResources().getString(R.string.aide_recherche));
        vuePourRecherche.setOnQueryTextListener(new SearchView.OnQueryTextListener() {

            /**
             * Méthode invoquée quand l'utilisateur valide la recherche,
```

```
             * i.e. quand il clique sur la loupe du clavier virtuel
             * @param query  texte tapé par l'utilisateur dans la zone de saisie
             * @return vrai si la recherche a pu être gérée
             */
            @Override
            public boolean onQueryTextSubmit(String query) {
                texte.setText(new StringBuilder(
                        getResources().getString(R.string.resultat_recherche)).append(query));
                return true;
            }

            /**
             * Méthode invoquée quand l'utilisateur modifie le texte de la recherche
             * @param s  texte modifie
             * @return vrai si le changement de texte a pu être géré
             */
            @Override
            public boolean onQueryTextChange(String s) {
                return true;
            }
        });

        return true;
    }


    /**
     * Appelée automatiqument chaque fois que l'utilisateur sélectionne une option
     * du menu d'options
     * @param item   option du menu sélectionnée par l'utilisateur
     * @return un booléen égal à vrai si l'option choisie a pu être correctement traitée
     */
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        /*
         * Un message de type Toast est affiché lorsque l'utilisateur
         * sélectionne une option du menu d'options
         */
        switch(item.getItemId()) {
            case R.id.premiere_option :
                Toast.makeText(this, R.string.choix_premiere_option,
                                Toast.LENGTH_LONG).show();
                break;
            case R.id.deuxieme_option :
                Toast.makeText(this, R.string.choix_deuxieme_option,
                        Toast.LENGTH_LONG).show();
                break;
            case R.id.troisieme_option :
                Toast.makeText(this, R.string.choix_troisieme_option,
                        Toast.LENGTH_LONG).show();
                break;
            case R.id.quatrieme_option :
                Toast.makeText(this, R.string.choix_quatrieme_option,
                        Toast.LENGTH_LONG).show();
                break;
            case R.id.cinquieme_option :
                Toast.makeText(this, R.string.choix_cinquieme_option,
                        Toast.LENGTH_LONG).show();
                break;
        }
        return true;
    }

}
```

Le cycle de vie d'une application

I) Le cycle de vie d'une application

Les applications Android ont un fonctionnement particulier. A tout moment, le système peut mettre en pause ou arrêter complètement une activité s'il le juge nécessaire. Par exemple : si l'application consomme trop de ressources, ou bien si un appel téléphonique est reçu ...

Plus précisément, chaque application fonctionne dans son propre processus. Le système Android est responsable de la création et de la destruction des processus, et gère ses ressources avec comme objectif la sécurité et aussi la disponibilité et la réactivité de l'appareil. Un processus peut donc être tué à tout moment par le système afin de libérer des ressources.

L'ordre d'arrêt des processus est déterminé par leur priorité. La priorité d'une application est égale à la priorité la plus élevée de ses différents composants. Deux applications avec la même priorité seront départagées par le temps qu'elles auront passé à une priorité plus faible.

Toutes les activités en cours sont placées dans une pile (gérée avec la politique LIFO). Lorsqu'une activité démarre, elle devient active et est placée au sommet de la pile. Si l'utilisateur retourne en arrière avec le bouton *back*, ou si l'activité courante est fermée pour une autre raison, la précédente dans la pile se retrouve au sommet et c'est celle-ci qui devient active.

Exemple illustrant l'enchaînement des processus

Voici le scénario : un utilisateur est sur l'écran d'accueil du terminal, et clique sur l'icône pour consulter ses courriels. Une fois la liste de courriels visible, il clique sur l'un d'eux pour le visualiser en détail. Il décide ensuite de consulter le site Internet pour lequel un lien apparaît dans le texte du courriel. Il clique donc sur le lien ce qui a pour effet de lancer le navigateur. De là, l'utilisateur clique sur un autre lien pour ouvrir l'application Google Maps. Ensuite, il retourne à sa consultation de messages.

Schématiquement, les étapes sont les suivantes :

- ✓ Ecran d'accueil
- ✓ Application messagerie présentant la liste courriels
- ✓ Application messagerie présentant le contenu d'un courriel précis
- ✓ Navigateur avec un premier site internet
- ✓ Navigateur avec l'application Google Maps

Que se passe-t-il au niveau des processus ? Les processus lancés sont successivement :

- ✓ Le processus système qui contient le gestionnaire d'activités et de la pile des appels utilisés par toutes les activités
- ✓ le processus *home* qui contient l'activité qui gère l'écran d'accueil
- ✓ lors du clic sur l'icône de messagerie, le système sauvegarde l'état de l'activité courante *home*
- ✓ le système crée le processus associé à la messagerie et lance cette activité
- ✓ lors du clic sur l'un des messages, l'état de l'activité messagerie est sauvegardé (juste l'état, et pas le contenu des courriels)
- ✓ le système lance l'activité qui affiche le détail du courriel sélectionné
- ✓ lors du clic sur un lien, l'état de l'activité courante est sauvegardé (cet état comporte le message en cours de lecture)
- ✓ un processus pour le navigateur est créé, et l'activité du navigateur est lancée
- ✓ l'utilisateur clique sur Google Maps, normalement l'état du navigateur devrait être sauvegardé. Mais supposons qu'il n'y ait plus de place en mémoire. Un processus doit donc être tué. Le processus *home* ne doit pas l'être, car il doit toujours être utilisable. Le navigateur ne sera pas tué non plus car il est récent. C'est l'application de lecture des messages qui sera détruite. Elle est détruite ainsi que son processus. Le processus Google Maps est créé et l'activité associée est lancée.
- ✓ Si l'utilisateur souhaite revenir en arrière, grâce à la touche *back*, l'application Google Maps disparaît du haut de la pile des activités. L'activité navigateur passe au premier plan ; elle était toujours en mémoire avec son processus
- ✓ Si l'utilisateur souhaite encore revenir en arrière pour voir son message, le système doit relancer le processus correspondant, et donc faire de la place en tuant le processus de l'application Google Maps. Un nouveau processus pour l'application de messagerie est lancé et une nouvelle instance d'activité de lecture de courriels est lancée, mais ne sera pas dans l'état laissé par l'utilisateur. L'activité de messagerie prend la place du navigateur en haut de la pile des activités.
- ✓ Si l'utilisateur souhaite revenir à la liste des courriels, le système n'aura pas besoin de créer un processus. Le processus de messagerie existe déjà. Il lui suffira de créer l'activité correspondante et de restaurer son état.

II) Le cycle de vie d'une activité

Une activité possède un cycle de vie. Celui-ci comporte différents états possibles. Les 3 principaux sont les suivants : **active** (active), **suspendue** (paused), et **arrêtée** (stopped).

Détaillons les états d'une activité :

- ✓ **activité active ou démarrée**: l'activité est au sommet de la pile, elle est visible et elle détient le focus de l'utilisateur, attend les entrées de celui-ci. Android tentera de la maintenir active le plus longtemps possible, si nécessaire en tuant les processus dédiés à des activités plus basses dans la pile. C'est l'appel à la méthode **onResume**, à la création ou à la reprise après l'état pause, qui permet à l'activité de passer dans cet état. Lorsque l'activité passera à l'état suspendu, la méthode ***onPause*** sera invoquée automatiquement

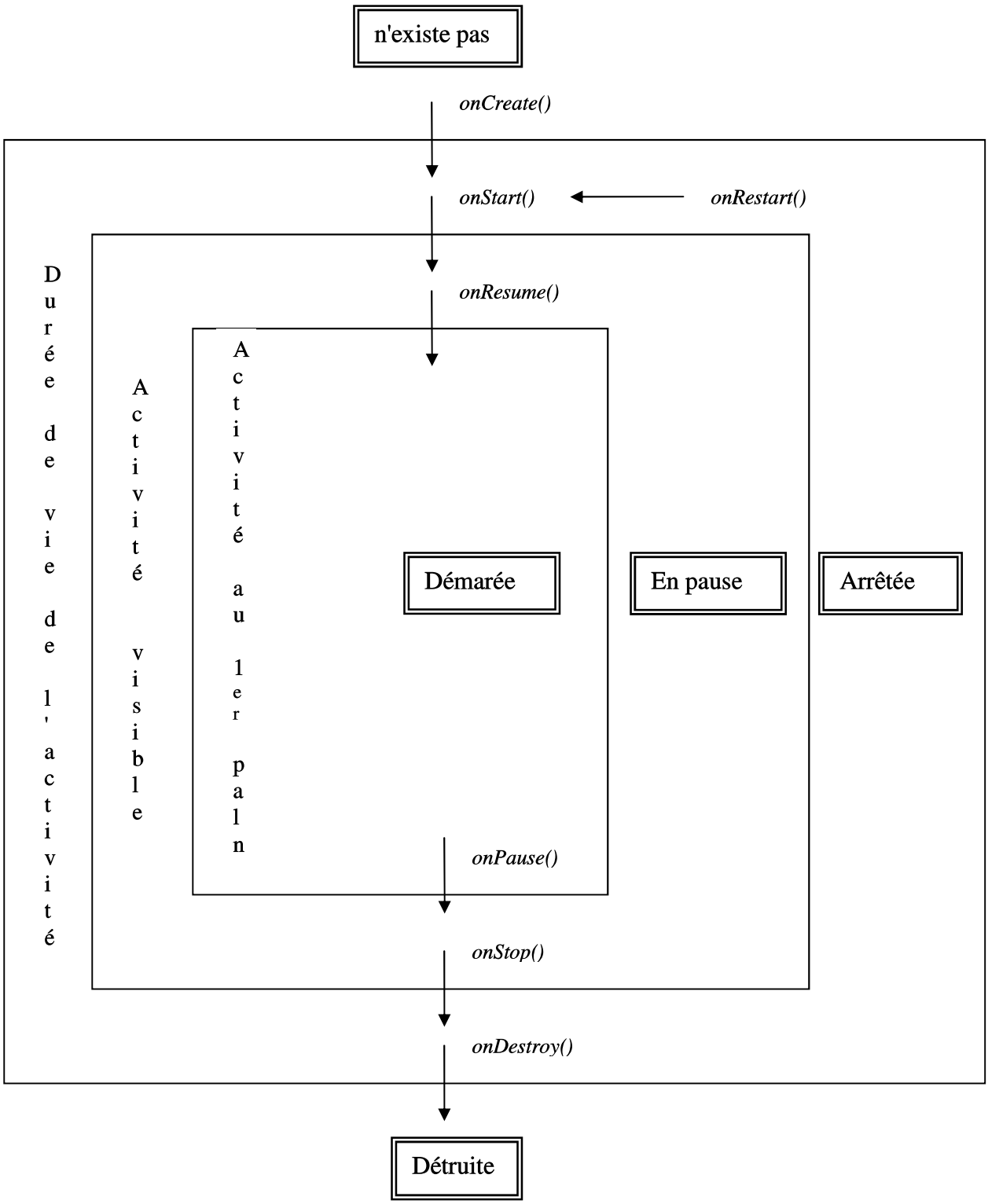
- ✓ **activité suspendue ou en pause** : l'activité est au moins en partie visible sur l'écran, mais elle ne détient plus le focus. Par exemple, un appel téléphonique est reçu et un message d'alerte apparaît sur l'écran au premier plan. Il masque donc l'activité suspendue. L'utilisateur ne peut pas interagir avec l'activité suspendue. Ce n'est que des les cas extrêmes de manque de ressources qu'Android tuera une activité en pause. Depuis, l'API 11, normalement il ne doit pas tuer une telle activité.

- ✓ **activité stoppée ou arrêtée** : l'activité n'est plus visible. Elle reste en mémoire avec toutes les informations décrivant son état. Elle est présente dans la pile d'exécution. Toutefois, Android peut décider de la tuer, s'il a besoin de ressources.

- ✓ **activité détruite** : l'activité n'existe plus. Elle a éventuellement été tuée à cause d'un manque de mémoire. Elle n'est plus dans la pile d'exécution.

En résumé, une activité dans l'état "en pause" ou "arrêtée" peut être détruite par Android. La probabilité qu'elle soit tuée est plus forte dans l'état "arrêtée" que dans l'état "en pause" (probabilité très faible dans l'état "en pause").

Ces changements d'états s'accompagnent d'appels automatiques à différentes méthodes de la classe ***Activity***. Plusieurs de ces méthodes devront être redéfinies dans les classes correspondant aux activités de l'application. Le schéma ci-dessous montre le cycle de vie d'une activité et les méthodes appelées lors des changements d'état.



Une classe qui hérite de ***Activity*** doit respecter le schéma suivant. Noter la présence dans chacune des méthodes, d'un appel à la méthode équivalente de la classe parente (mot-clé **super**). Cet appel est obligatoire.

```
public class MonActivite extends Activiy {

    /**
     * Appelée lorsque l'activité est créée.
     * Permet de restaurer l'état de l'interface utilisateur. Plus précisément :
     *  - lorsque l'activité est lancée pour la première fois, le paramètre est égal à null
     *  - les fois suivantes : le paramètre contient l'instance de type Bundle fabriquée par la
     *     méthode onSaveInstanceState
     */
    @Override
    public void onCreate(Bundle saveInstanceState) {
       super.onCreate(savedInstanceState);
       // initialiser l'activité (désérialiser l'interface graphique, la configurer, lancer
       // en arrière plan les tâches qui doivent s'exécuter pendant toute la durée de vie de
       // l'application ...)
    }


    /**
     * Appelée lorsque l'activité démarre.
     * On pourra initialiser ici les contrôles
     */
    @Override
    public void onStart() {
       super.onStart();
       // placer le code ici
    }


    /**
     * Appelée lorsque l'activité sort de son état de veille
     * On sait que l'activité a déjà été visible dans ce processus
     */
    @Override
    public void onRestart() {
       super.onRestart();
       // placer le code ici
    }


    /**
     * Appelée après le démarrage ou une pause, juste avant que l'activité ne passe au premier plan.
     * Il faut relancer les opérations arrêtées (les threads, les animations), rafraichir l'interface
     * initier l'accès à des ressources... On peut aussi reconstruire l'interface graphique
     * en fonction de ce qui s'est passé depuis que l'utilisateur l'a vue pour la dernière fois
     * (par exemple : actualiser un flux RSS, afficher l'heure ...).
     * Les instructions placées dans cette méthode doivent s'exécuter rapidement.
     */
    @Override
    public void onResume() {
       super.onResume();
       // placer le code ici
    }


    /**
     * Appelée lorsque l'activité est suspendue, c'est-à-dire passe en arrière plan.
     * Il se peut qu'elle soit ensuite arrêtée.
     * On peut arrêter les actions qui consomment des ressources, et notamment de la batterie
     * (par exemple, arrêter une animation puisqu'elle ne sera pas visible, libérer des ressources
     * comme le GPS, la caméra ...)
     * On évite toutefois d'effectuer des tâches longues, ceci dans le but de fluidifier
     * l'actualisation de l'interface (par exemple, on ne sauvegarde pas nécessairement ici, les
     * données dans une base de données). Il faut parfois faire un compromis entre les actions
     * à placer dans onPause, et celles de onStop.
     */
```

```
    @Override
    public void onPause() {
       // placer le code ici
       super.onPause();
    }


    /**
     * Appelée lorsque l'activité devient invisible.
     * On peut libérer les écouteurs, arrêter les threads...,
     * en fait, suspendre tous les traitements inutiles lorsque l'activité n'est pas visible.
     * Les opérations réalisées ici viennent compléter celles de la méthode onPause().
     * On effectue dans onPause les opérations longues, telles que l'écriture dans une base de
     * données.
     */
    @Override
    public void onStop() {
       // placer le code ici
       super.onStop();
    }


    /**
     * Appelée lorsque l'activité a fini son cycle de vie, plus précisément
     *   soit quand l'activité elle-même décidera de se détruire, grâce à un appel à la méthode
     *      finish,
     *   soit si Android a besoin de mémoire et a fermé prématurément l'activité. Cependant,
     *      il y a un risque faible que onDestroy ne soit pas appelée, si le besoin en ressource
     *      est très urgent.
     *   soit si l'utilisateur appuie sur le bouton "back" et qu'il n'y a pas d'activité précédente
     *     au sein de l'application courante (c'est le cas notamment si l'application contient
     *     une seule activité).
     */
    @Override
    public void onDestroy() {
       // placer le code ici (libération de mémoire, fermeture de
       // fichiers, de connexion à une base de données et autres opérations de "nettoyage"...)
       super.onDestroy();
    }


    /**
     * Appelée lorsque l'activité termine son cycle visible.
     * On sauvegarde les données importantes.
     */
    @Override
    public void onSaveInstanceState(Bundle saveInstanceState) {
       // placer le code ici
       super.onSaveInstanceState(saveInstanceState);
    }


   /**
     * Appelée après onCreate.
     * Ne sera appelée que si l'activité a été tuée par le système
     * depuis qu'elle a été visible pour la dernière fois
     * Les données sont rechargées et l'interface utilisateur est restaurée
     * dans le bon état
     */
    @Override
    public void onRestoreInstanceState(Bundle saveInstanceState) {
       super.onRestoreInstanceState(saveInstanceState);
       // placer le code ici
    }
```

Si lors de l'exécution de l'activité, l'utilisateur retourne à l'écran d'accueil (home), l'activité passera dans l'état "arrêtée". Si l'utilisateur reprend ensuite l'activité, la première méthode invoquée sera *onRestart* puis *onStart*.

Si par contre c'est Android qui arrête l'exécution de l'activité, et si l'activité est ensuite relancée, c'est la méthode *onCreate* qui sera invoquée en premier.

Méthodes *onCreate* et *onDestroy*

Lorsque l'activité est lancée pour la première fois, le paramètre de ***onCreate*** est égal à *null*. Si l'activité a été tuée et qu'elle est relancée par la suite, le paramètre de ***onCreate*** sera le *Bundle* fabriqué par la méthode ***onSaveInstanceState***().

C'est dans ***onCreate*** que l'on désérialise l'interface utilisateur, on la configure, et on lance des tâches en arrière plan (les services et les threads, voir plus loin dans le cours), celles qui doivent s'exécuter pendant toute la durée de vie de l'application.

La méthode ***onDestroy*** sera appelée :

- ✓ soit quand l'activité elle-même décidera de se détruire grâce à un appel à la méthode ***finish***. L'appel à la méthode ***finish*** figure dans le code de l'activité.
- ✓ soit si Android a besoin de mémoire et a fermé prématurément l'activité. Cependant, il y a un risque faible que ***onDestroy*** ne soit pas appelée si le besoin en ressource est très urgent.
- ✓ soit si l'utilisateur appuie sur le bouton "*back*" et qu'il n'y a pas d'activité précédente au sein de l'application courante. C'est le cas notamment si l'application contient une seule activité.

On libère dans cette méthode, les ressources allouées par ***onCreate***. On ferme donc les connexions réseau, ou avec une base de données.

Méthodes *onStart, onRestart, onStop*

La méthode ***onRestart*** n'est invoquée que lorsque l'activité a été stoppée et redémarre.

C'est dans la méthode ***onStop*** que l'on peut mettre en pause ou stopper les animations (puisqu'elles ne seront plus visibles), les *threads*, les écouteurs sur les capteurs, les recherches GPS ou d'une manière générale tous les traitements destinés à mettre à jour l'interface utilisateur. On peut aussi libérer les objets de type curseur en lien avec la base de données.

Dans la méthode ***onStart***, il faudra donc relancer les traitements arrêtés dans la méthode ***onStop***. Plus précisément, le redémarrage de ces traitements sera réparti entre ***onStart*** et ***onRestart***.

Méthodes *onPause* et *onResume*

D'une manière générale, il faut que le code des méthodes ***onPause*** et ***onResume*** soit rapide à exécuter. Le but est que l'application soit réactive lorsqu'elle passe au premier plan, ou au contraire en arrière plan.

La méthode ***onPause*** est invoquée lorsque l'activité passe en arrière plan. On peut donc arrêter ici des animations (puisque non visibles et consommatrices de ressources), libérer des ressources (GPS, caméra, ...) et tous les traitements qui sollicitent beaucoup le processeur. Il faut répartir judicieusement ces libérations entre les méthodes ***onPause*** et ***onStop***. Les deux méthodes sont complémentaires. Comme dit précédemment, l'exécution de ***onPause*** doit être rapide.

On évite de sauvegarder ici les données dans la base de données, car c'est une tâche longue qui porterait préjudice à la réactivité de l'application.

La méthode ***onResume*** est appelée immédiatement avant que l'activité ne passe au premier plan. On peut lancer ici des *threads*, des animations, initier l'accès à des ressources, en fait relancer les traitements suspendus dans ***onPause***. On doit aussi reconstruire l'interface graphique en fonction de ce qui s'est passé depuis que l'utilisateur l'a vue depuis la dernière fois. Par exemple : actualiser un flux RSS, afficher l'heure .... Les instructions placées ici doivent s'exécuter rapidement.

Méthodes *onSaveInstanceState* et *onRestoreInstanceState*

Quand l'application est quittée de manière normale, par exemple si l'utilisateur appuie sur *back*, ou si un appel à *finish* est effectué dans une activité, Android ne garde pas d'informations en mémoire sur l'état de l'activité ou de l'application.

Par contre, si c'est Android qui a dû détruire l'activité (manque de ressource ou changement d'orientation de l'écran), alors il va garder en mémoire tout ce qui est nécessaire pour restaurer l'état de l'activité, dans le cas où celle-ci serait relancée. Concrètement, au prochain lancement, le paramètre *Bundle* de ***onCreate*** sera renseigné avec les informations enregistrées sur l'état de l'activité. Si ces informations sauvées par défaut ne sont pas suffisantes, le programmeur devra coder lui-même l'enregistrement des informations complémentaires, dans la méthode ***onSaveInstanceState.***

Plus précisément, la sauvegarde de l'état d'une instance est gérée par ***onSaveInstanceState***. L'implémentation prédéfinie de cette méthode sauvegarde l'état a priori modifiable des éléments de l'interface. C'est-à-dire tous les *widgets* qui possèdent un identifiant (il s'agit de l'identifiant défini, dans le fichier *XML*, via la propriété *android:id*). Le programmeur peut rajouter des instructions pour sauvegarder d'autres éléments.

L'état de l'interface peut ensuite être récupéré dans les appels à ***onRestoreInstanceState*** et à ***onCreate***, au choix du programmeur. L'appel automatique à ***onRestoreInstanceState*** se fait entre les appels à *onStart* et *onResume*.

Un objet de type ***Bundle*** est l'équivalent d'une table de hachage qui à une chaîne de caractères associe un élément. On peut placer dans le *Bundle* uniquement des objets sérialisables.

Dans la méthode ***onSaveInstanceState,*** pour enregistrer des informations dans le ***Bundle***, on utilise les mêmes méthodes que pour enregistrer des préférences (voir plus loin dans le cours). Par exemple, pour enregistrer une chaîne de caractères, on écrira :

```
saveInstanceState.putString(CLE_CHAINE, "exemple de chaîne");
```

où *saveInstanceState* est le paramètre de ***onSaveInstanceStat*** et *CLE_CHAINE* est une constante de type *String* contenant la clé de la valeur à enregistrer.

Ensuite dans la méthode ***onRestoreInstanceState*** ou ***onCreate***, on souhaitera récupérer la valeur de l'information ayant pour clé *CLE_CHAINE*. On procèdera ainsi :

```
if (savedInstanceState != null && savedInstanceState.containsKey(CLE_CHAINE)) {
    texte = savedInstanceState.getString(CLE_CHAINE)
}
```

Remarques

Si une activité invoque la méthode ***finish***, la méthode ***onDestroy*** est appelée immédiatement, donc ***onPause*** et ***onStop*** ne le seront pas.

Si dans l'activité, une exception levée n'est pas interceptée (donc dans un bloc *catch*), l'application sera détruite dans passer par ***onDestroy***.

Gérer la persistance des données
Les préférences - Les fichiers

Dans ce chapitre, nous envisageons deux techniques pour sauvegarder et charger des données à partir d'un support mémoire permanent, ou autrement dit deux techniques pour gérer la persistance des données:

- ✓ Les **préférences** partagées sont un moyen simple de stocker des informations sous la forme de paires clé/valeur. Ces informations pourront ensuite être consultées lors du prochain lancement de l'application. Le plus souvent ces paires serviront à mémoriser les préférences de l'utilisateur, mais ce n'est pas leur seul usage.

- ✓ Tout comme dans le langage Java standard, il est possible de stocker des informations dans des **fichiers** grâce aux classes du package *java.io*.

I) Les préférences

Il s'agit d'une technique "légère" pour stocker des informations sous la forme de paires *nom-valeur* ou *clé-valeur*, dans des fichiers au format XML. Les valeurs de ces paires peuvent être de type : *boolean*, *String*, *float*, *long* et *integer*. Les paires peuvent servir à stocker des valeurs par défaut, des attributs, des préférences ou l'état de l'interface utilisateur, par exemple.

Comment accéder au fichier des préférences ?

Pour créer un ensemble de paires *nom-valeur* qui persistent entre les différentes exécutions d'une application et qui sont partagées entre les composants qui constituent l'application, on utilise la classe ***SharedPreferences***.

Pour créer, ou modifier une préférence partagée, on invoque la méthode ***getSharedPreferences,*** dans le contexte courant, en lui donnant en paramètre le nom du fichier contenant les préférences (type *String*), et le mode d'accès (en général privé).

Le profil de la méthode est le suivant :

SharedPreferences ***getSharedPreferences****(String nomFichierPreference, int modeAcces)*

Elle renvoie le contenu du fichier de préférences dont le nom est donné en argument. Le contenu est placé dans une instance de type ***SharedPreferences***. La méthode renvoie *null* si le fichier n'est pas trouvé.

Dans l'exemple ci-dessous, on suppose que le nom du fichier contenant les préférences est « *monFichierPref.xml* » :

```
SharedPreferences mesPreferences =
                getSharedPreferences("monFichierPref.xml", Activity.MODE_PRIVATE);
```

Le mode d'accès peut être :

MODE_PRIVATE	c'est le mode par défaut (et aussi le plus courant), le fichier ne sera accessible que par l'application qui l'a créé
MODE_WORLD_READABLE	les autres applications pourront lire le fichier, mais pas le modifier
MODE_WORLD_WRITABLE	les autres applications pourront modifier le fichier
MODE_MULTI_PROCESS	pour autoriser plusieurs processus à modifier le fichier

Un fichier de préférences par défaut est associé à chaque application. Si dans une application donnée, on utilise un seul ensemble de préférences, le plus simple est d'utiliser le fichier des préférences par défaut. On évite ainsi d'indiquer un nom de fichier précis lors de l'accès à celui-ci :

```
SharedPreferences mesPreferences =
            PreferenceManager.getDefaultSharedPreferences(getApplicationContext());
```

Consulter des préférences partagées

Pour consulter les données sauvegardées dans un fichier de préférences, on utilise la méthode ***getXXX,*** *où XXX* précise un type. Cette méthode prend en paramètre une clé et une valeur. Cette valeur est la valeur par défaut qui est renvoyée si aucune valeur n'a été associée, pour l'instant, à cette clé. C'est aussi un moyen d'initialiser la valeur associée à une clé.

```
boolean modeSilencieux = mesPreferences.getBoolean("silencieux", false);
String chaineLibelle = mesPreferences.getString("libelle", "Libellé : ");
```

Ou bien, on peut aussi récupérer simultanément toutes les paires *clé-valeur* dans une ***Map*** :

```
Map<String, ?> toutesLesPreferences = mesPreferences.getAll();
```

Pour seulement vérifier l'existence d'une paire clé/valeur précise, on teste la présence de la clé en invoquant la méthode ***contains*** :

```
if (mesPreferences.contains("libelle")) {
      . . .
}
```

Modifier des préférences partagées

Pour modifier une préférence partagée, il faut passer par l'intermédiaire d'un éditeur de préférences comme indiqué ci-dessous (noter l'appel à la méthode ***edit*** sur l'instance de type ***SharedPreferences***) :

```
SharedPreferences.Editor editeur = mesPreferences.edit();
editeur.putBoolean("silencieux", true);
editeur.putString("libelle", "Titre :");
editeur.putFloat("valeur", 34.5f);
editeur.putInt("quantite", 10);
editeur.putLong("grandequantite", 1000);
```

Il existe une méthode d'écriture ***putXXX*** par type primitif. On passe en paramètre de celle-ci la clé et la valeur à associer à cette clé. La méthode renvoie l'instance `SharedPreferences.Editor` modifiée. On peut donc chaîner les appels les uns aux autres :

```
editeur.putBoolean("silencieux", true).putString("libelle", "Titre :");
```

Pour supprimer une préférence, on invoque la méthode ***remove***. Par exemple :

```
editeur.remove("quantite");
```

Pour supprimer toutes les préférences, on utilise la méthode ***clear***.

Puis pour sauvegarder de manière asynchrone ou synchrone les modifications, il faut invoquer respectivement ***apply*** ou ***commit***, sur l'instance éditeur :

```
editeur.apply();          ou          editeur.commit();
```

commit est une opération synchrone et renvoie un booléen (vrai si opération réussie), alors que ***apply*** ne renvoie pas de résultat et est asynchrone.

Remarques

- Les préférences internes sont stockées dans le fichier situé à l'emplacement :
  `/data/data/nom_du_package/shared_prefs/lefichier.xml`

- Pour faciliter la tâche du développeur, la plate-forme Android fournit un type d'activité pour présenter un écran de saisie des préférences (analogue à celui des applications natives d'Android) et gérer leur persistance. Cette activité hérite de ***Activity*** et se nomme ***PreferenceActivity***.

II) Les fichiers

Les applications vont pouvoir créer, consulter, modifier ou télécharger des fichiers qui leur sont propre. La manipulation des fichiers ressemble à ce qui se fait en Java standard, avec quelques différences toutefois. Les fichiers manipulés seront stockés :

- ✓ soit sur l'espace de **stockage interne à l'application,** c'est-à-dire en mémoire permanente (c'est celle qui contient les fichiers système, la plupart des applications). Sur **le stockage interne**, la présence des fichiers dépend de celle de l'application. Donc les fichiers seront supprimés si l'utilisateur désinstalle l'application. Cependant, comme la mémoire interne peut être de taille réduite sur certains terminaux, on évite en général de placer là les gros fichiers.

- ✓ soit sur l'espace de **stockage externe**. Il s'agit ici d'un stockage partagé, accessible par toutes les applications. Son avantage est d'offrir plus d'espace. Son inconvénient est que n'importe quelle application peut y accéder pour modifier ou supprimer des fichiers. D'un point de vue matériel, cet espace de stockage peut se situer sur une carte SD, ou bien sur une partition séparée de l'espace de stockage interne. Les fichiers stockés sur le support externe ne sont pas obligatoirement disponibles: par exemple, la carte SD peut avoir été retirée, ou le terminal être monté en vue d'un accès depuis un ordinateur.

Attention toutefois, un fichier situé sur le stockage interne pourrait lui aussi être supprimé par un utilisateur qui s'octroierait les droits du superutilisateur, ou *root*.

Remarque importante : les opérations d'accès aux fichiers étant longues, il est fortement conseillé de les effectuer dans des *threads* dédiés, donc différents du *thread* UI, celui qui gère l'interface utilisateur. Cette remarque s'applique également pour les accès aux préférences. La notion de *thread* sera étudiée dans un prochain chapitre.

2.1) Les fichiers situés sur l'espace de stockage interne

Il existe 2 catégories de fichiers situés sur le stockage interne :

- Les fichiers définis en tant que ressource de l'application, ce sont des fichiers statiques
- Les autres fichiers qui eux ne sont pas obligatoirement statiques dans le sens où l'application peut modifier leur contenu

2.1.a) Fichiers définis en tant que ressource de l'application

Il est possible d'embarquer des fichiers dans l'application elle-même, comme par exemple, les règles d'un jeu, un dictionnaire de mots .... Ces fichiers seront alors directement empaquetés dans le fichier *.apk*

de l'application. Pour ce faire, les fichiers doivent être rangés dans le dossier `res/raw` du projet. Si `mon_fichier` est un fichier dans ce dossier, dans le code Java, on y accède de la manière suivante :

```
Resources res = getResources();
InputStream flux = res.openRawRessource(R.raw.mon_fichier);
```

On ne pourra pas modifier ce fichier dans l'application. Cette approche convient donc pour stocker les données statiques, ou, par exemple, des données initiales qui serviront à initialiser une base de données.

2.1.b) Fichiers placés dans le stockage interne de l'application

Pour accéder à un fichier présent dans le stockage interne, la première étape consiste à faire appel à l'une des méthodes suivantes, définies dans la classe ***Context*** :

FileInputStream openFileInput (String nomFichier)
FileOutputStream openFileOutput (String nomFichier, int mode)

Ces méthodes attendent en paramètre un nom de fichier, et pas un chemin d'accès au fichier. En effet, les fichiers internes sont tous stockés à un emplacement bien précis :

```
data/data/ nom du package de l'application / files
```

Le deuxième argument de ***openFileOutput*** doit préciser le mode d'accès au fichier. Il a pour valeur une constante de la classe ***Context***. Celle-ci peut être :

`MODE_PRIVATE`	c'est le mode par défaut et le plus courant. Il signifie "privé" à l'application.
`MODE_WORLD_READABLE`, `MODE_WORLD_WRITABLE`, ou `MODE_APPEND`	on peut utiliser l'opérateur **|** pour associer ce dernier mode à un autre.

Notons que si l'on souhaite que des informations soient partagées entre plusieurs applications, il sera préférable d'utiliser la notion de fournisseur de contenu intégrée à Android.

Ensuite, l'accès au fichier est analogue au langage Java standard :

- ✓ on enveloppe ces flux selon les besoins dans un ***InputStreamReader*** ou un ***OutputStreamReader***. Eventuellement, on rajoute un ***BufferedReader*** si le fichier à lire est un fichier texte. En effet, à la base ces fichiers sont des fichiers binaires et pas des fichiers texte.
- ✓ on lit ou on écrit les données dans le fichier
- ✓ on ferme le flux avec un appel à la méthode ***close()***

Ecriture dans le fichier

Il faut écrire dans le fichier des données « brutes », c'est-à-dire directement la valeur d'un octet ou d'un tableau d'octets. Les deux méthodes d'écriture principales sont :

void write(int oneByte) et ***void write(byte[] buffer)***

Remarques

☐ Deux processus qui essaient de lire en même temps un même fichier accèderont chacun à leur propre version du fichier. Si on souhaite qu'un même fichier soit accessible à partir de plusieurs endroits et que les accès concurrents soient gérés, il faut utiliser la notion de fournisseur de contenu (*content provider*).

☐ Il ne peut pas exister de sous-répertoire dans la zone interne.

☐ On peut aussi effectuer des opérations plus globales sur un fichier, en utilisant les méthodes suivantes définies dans la classe ***Context*** :

- ✓ supprimer un fichier avec — ***boolean deleteFile(String name)***
- ✓ récupérer la liste des fichiers présents — ***String[ ] fileList()***
- ✓ obtenir l'emplacement du stockage interne — ***File getFilesDir()***
- ✓ *etc ...*

Exemple : lecture d'un fichier et présentation dans une liste

On souhaite afficher la liste des pays de l'Union Européenne. Les noms des pays sont stockés dans un fichier, situé en mémoire interne.

Ce fichier est un fichier texte. Chaque ligne du fichier contient un nom de pays. Le nom du fichier est *fichierpays.txt*.

Remarque : la lecture du fichier est effectuée dans le *thread* UI, car pour l'instant la notion de *thread* n'a pas été étudiée. Il faudrait améliorer cet exemple en plaçant le code qui réalise l'accès au fichier dans un *thread* dédié (voir un prochain chapitre du cours).


```
/*
 * Présente la liste des pays de l'union européenne
 * ActivitePaysUE.java                                    02/15
 */
package com.example.exfichier;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
.....

/**
 * Cette activité présente à l'utilisateur la liste des pays de l'Union Européenne,
 * sous la forme d'une liste.
 * Les pays de l'UE sont récupérés à partir d'un fichier, situé dans la mémoire interne
 * du terminal.
 * @author C. Servières
 * @version 1.0
 */
```

```
public class ActivitePaysUE extends ListActivity {

    /** Nom du fichier contenant les pays de l'union européenne */
    private static final String NOM_FICHIER = "fichierpays.txt";

    /**
     * Tag utilisé dans les messages de log. Les messages de log sont affichés en cas
     * de problème lors de l'accès au fichier
     */
    private static final String TAG = "PaysUE";

    /** Label avec message de bienvenue */
    private TextView labelBienvenue;

    /** Liste des pays présentés par l'application */
    private  ArrayList<String> lesPays;;


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activite_pays_ue);

        // on récupère la référence sur le TextView
        labelBienvenue = (TextView) findViewById(R.id.LabelBienvenue);

        /*
         *  on constitue la liste des pays en consultant le fichier texte.
         *  On lit un par un les pays présents dans le fichier et on les
         *  ajoute à la liste.
         */
        lesPays = new ArrayList<>();
        String paysLu;
        try {
            InputStreamReader fichier =
                        new InputStreamReader(openFileInput(NOM_FICHIER));
            BufferedReader fichierTexte = new BufferedReader(fichier);

            while ( (paysLu = fichierTexte.readLine())!= null ) {
                  lesPays.add(paysLu);
            }
            fichier.close();
        } catch (FileNotFoundException e) {
            Log.e(TAG, "Le fichier " + NOM_FICHIER + " n'existe pas.");
        } catch (IOException e) {
            Log.e(TAG, "Problème de lecture dans le fichier " + NOM_FICHIER);
        }

        // On crée un adaptateur pour rassembler les données à afficher
        ArrayAdapter<String> adaptateur =
                new ArrayAdapter<String>(this,
                                         android.R.layout.simple_list_item_1,
                                         lesPays);
        setListAdapter(adaptateur);

        // on adapte le message du label de bienvenue (indiquer le nombre de pays)
        labelBienvenue.append(" " + lesPays.size());
    }
}
```

Exemple d'écriture d'une chaîne de caractères

Dans cet exemple, on souhaite écrire une seule ligne de texte dans un fichier texte. Le nom du fichier texte est *mon_fichier.txt*. Un appel à la méthode ***getBytes*** est effectué pour encoder la chaîne de caractères à écrire en binaire. C'est l'encodage en binaire de la chaîne qui est effectivement écrit dans le fichier.

```
String uneChaine = "Une ligne du fichier texte";  // texte à écrire dans le fichier

FileOutputStream fichier = openFileOutput("mon_fichier.txt", Context.MODE_PRIVATE);

// écriture, dans le fichier, de la chaîne encodée en binaire
fichier.write(uneChaine.getBytes());

fichier.close();
```

2.2) Les fichiers situés sur l'espace de stockage externe

Sécurité

Pour accéder à un fichier placé en stockage externe, il faut ajouter une permission dans le fichier ***manifest.xml*** :

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

Note : ces permissions sont obligatoires même si le fichier est situé dans le stockage interne, avec les versions d'Android antérieures à KitKat.

Vérifier l'accessibilité du fichier

Comme indiqué précédemment, les fichiers externes peuvent résider sur une carte mémoire amovible, une carte SD, ou micro SD, sur une mémoire flash supplémentaire, ou encore une partition de la mémoire du terminal... En résumé, ce stockage n'est pas toujours disponible physiquement. Il faut donc d'abord vérifier sa disponibilité, en utilisant les méthodes ou constantes de la classe ***Environment***.

Pour obtenir l'état du stockage externe, on utilise la méthode :

String Environment.getExternalStorageState( )

La chaîne renvoyée est une constante de type *String* telle que, par exemple :

Environment.MEDIA_MOUNTED	stockage disponible en lecture/écriture
Environment.MEDIA_REMOVED	le support externe n'est pas présent
Environment.MEDIA_MOUNTED_READ_ONLY	(utiliser *equals* pour tester !)
....	

Le stockage externe peut gérer des fichiers privés (à l'application) ou bien partagés. Les fichiers privés seront détruits automatiquement si l'application est désinstallée. Ce n'est pas le cas des fichiers partagés, puisqu'ils peuvent être utilisés par d'autres applications.

Chaque application dispose de son propre répertoire dans la zone de stockage externe de l'appareil. Ce répertoire est

```
répertoire racine stockage externe/Android/data/nom du package de l'application/files
```

Ce répertoire peut contenir des sous-répertoires avec des noms système prédéfinis pour le stockage des différents types de fichiers. Par exemple : ***Movies*** pour film, ***DCIM*** pour images et photos, ***Alarms*** pour les alarmes, ***Download*** pour les téléchargements ...

De plus, comme dit précédemment, ce répertoire est automatiquement détruit lors de la désinstallation de l'application. Il est donc conseillé de placer dans celui-ci seulement les fichiers externes privés à l'application.

2.2.a) Fichiers privés

Pour accéder aux fichiers privés (sur le stockage externe), on utilise la méthode suivante de la classe ***Context*** :

File getExternalFilesDir(String catégorie)

La méthode renvoie une instance qui correspond au répertoire spécifié en argument. Le paramètre *catégorie* peut être égal à ***null*** (dans ce cas c'est le répertoire racine qui est renvoyé) ou bien à une constante chaîne de caractères parmi la liste suivante. Ces constantes sont définies dans la classe ***Environment*** :

```
DIRECTORY_MUSIC                 DIRECTORY_PODCASTS
DIRECTORY_RINGTONES             DIRECTORY_ALARMS
DIRECTORY_NOTIFICATIONS         DIRECTORY_PICTURES
DIRECTORY_MOVIES                ...
```

Exemple

Pour obtenir une référence sur un sous-ensemble du répertoire de stockage externe, plus précisément celui qui contient les fichiers images ou photos, on écrit :

```
File outFile = Environment.getExternalFilesDir(Environment.DIRECTORY_DCIM);
```

Remarques

□ Une fois l'instance de type ***File*** obtenue, on peut utiliser les classes et les méthodes du langage Java standard pour effectuer des accès aux fichiers du répertoire.

□ Contrairement au stockage interne, il est possible de créer, dans le répertoire racine (et dans ses sous-répertoires) du stockage externe, ses propres sous-répertoires en invoquant la méthode du Java standard : ***boolean File.mkdirs()***

2.2.b) Fichiers partagés

Les fichiers partagés sont ceux présents dans un répertoire accessible par l'utilisateur du terminal, et par toutes les autres applications. Ils ne seront pas supprimés lors de la désinstallation de l'application, puisqu'ils sont potentiellement utilisables par d'autres applications.

L'accès aux fichiers externes partagés se fait via la classe ***Environment***, et en appelant la méthode :

File getExternalStoragePublicDirectory(String catégorie)

L'argument *catégorie* peut avoir les mêmes valeurs que pour l'accès aux fichiers privés.

2.3) Les fichiers situés dans le cache de l'application

Un emplacement spécifique est réservé à chaque application pour qu'elle puisse stocker des fichiers temporaires. Plus précisément, un emplacement est réservé sur le stockage interne et un autre sur le stockage externe. Le programmeur peut utiliser l'un ou l'autre, sachant que le volume du stockage externe est plus grand que celui de l'interne.

La zone de cache associée à l'application est supprimée automatiquement lorsque l'application est désinstallée. Notons également que si le système Android ne dispose plus de suffisamment d'espace en mémoire, il peut supprimer les fichiers de la mémoire cache interne. D'une manière générale, il faut veiller à ce que l'application supprime les fichiers temporaires dont elle n'a plus besoin.

Pour accéder aux zones de cache, on invoque, à partir du contexte courant, l'une ou l'autre des méthodes suivantes :
getCacheDir() ou ***getExternalCacheDir()***.

Sécurité

Pour accéder en écriture au cache du support externe, il faut ajouter la permission adéquate dans le fichier ***manifest.xml*** :

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