Quatrième partie

Concepts avancés

18. Le travail en arrière-plan

L'un de vos objectifs prioritaires sera de travailler sur la réactivité de vos applications, c'est-à-dire de faire en sorte qu'elles ne semblent pas molles ou ne se bloquent pas sans raison apparente pendant une durée significative. En effet, l'utilisateur est capable de détecter un ralentissement s'il dure plus de 100 ms, ce qui est un laps de temps très court. Pis encore, Android lui-même peut déceler quand votre application n'est pas assez réactive, auquel cas il lancera une boîte de dialogue qui s'appelle ANR et qui incitera l'utilisateur à quitter l'application. Il existe deux évènements qui peuvent lancer des ANR :

1. L'application ne répond pas à une impulsion de l'utilisateur sur l'interface graphique en moins de cinq secondes.
2. Un `Broadcast Receiver` s'exécute en plus de dix secondes.

C'est pourquoi nous allons voir ici comment faire exécuter du travail en arrière-plan, de façon à exécuter du code en parallèle de votre interface graphique, pour ne pas la bloquer quand on veut effectuer de grosses opérations qui risqueraient d'affecter l'expérience de l'utilisateur.

18.1. La gestion du multitâche par Android

Comme vous le savez, un programme informatique est constitué d'instructions destinées au processeur. Ces instructions sont présentées sous la forme d'un code, et lors de l'exécution de ce code, les instructions sont traitées par le processeur dans un ordre précis.

Tous les programmes Android s'exécutent dans ce qu'on appelle un **processus**. On peut définir un processus comme une instance d'un programme informatique qui est en cours d'exécution. Il contient non seulement le code du programme, mais aussi des variables qui représentent son état courant. Parmi ces variables s'en trouvent certaines qui permettent de définir la plage mémoire qui est mise à la disposition du processus.

Pour être exact, ce n'est pas le processus en lui-même qui va exécuter le code, mais l'un de ses constituants. Les constituants destinés à exécuter le code s'appellent des **threads** (« fils d'exécution » en français). Dans le cas d'Android, les threads sont contenus dans les processus. Un processus peut avoir un ou plusieurs threads, par conséquent un processus peut exécuter plusieurs portions du code en parallèle s'il a plusieurs threads. Comme un processus n'a qu'une plage mémoire, alors tous les threads se partagent les accès à cette plage mémoire. On peut voir à la figure suivante deux processus. Le premier possède deux threads, le second en possède un seul. On peut voir qu'il est possible de communiquer entre les threads ainsi qu'entre les processus.

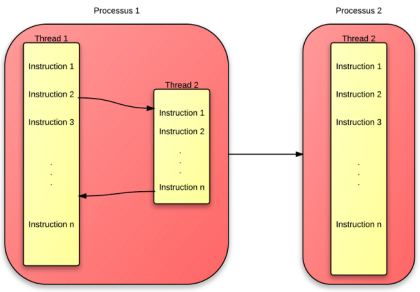

FIGURE 18.1. – Schéma de fonctionnement des threads

Vous vous rappelez qu'une application Android est constituée de composants, n'est-ce pas ? Nous n'en connaissons que deux types pour l'instant, les activités et les receivers. Il peut y avoir plusieurs de ces composants dans une application. Dès qu'un composant est lancé (par exemple au démarrage de l'application ou quand on reçoit un `Broadcast Intent` dans un receiver), si cette application n'a pas de processus fonctionnel, alors un nouveau sera créé. Tout processus nouvellement créé ne possède qu'un thread. Ce thread s'appelle le **thread principal**.

En revanche, si un composant démarre alors qu'il y a déjà un processus pour cette application, alors le composant se lancera dans le processus en utilisant le même thread.

18.1.1. Processus

Par défaut, tous les composants d'une même application se lancent dans le même processus, et d'ailleurs c'est suffisant pour la majorité des applications. Cependant, si vous le désirez et si vous avez une raison bien précise de le faire, il est possible de définir dans quel processus doit se trouver tel composant de telle application à l'aide de la déclaration du composant dans le Manifest. En effet, l'attribut `android:process` permet de définir le processus dans lequel ce composant est censé s'exécuter, afin que ce composant ne suive pas le même cycle de vie que le restant de l'application. De plus, si vous souhaitez qu'un composant s'exécute dans un processus différent mais reste privé à votre application, alors rajoutez « `:` » à la déclaration du nom du processus.

```
```

Ici, j'ai un receiver qui va s'enclencher dès que la batterie devient faible. Configuré de cette manière, mon receiver ne pourra démarrer que si l'application est lancée (comme j'ai rajouté « `:` », seule mon application pourra le lancer) ; cependant, si l'utilisateur ferme l'application alors que le receiver est en route, le receiver ne s'éteindra pas puisqu'il se trouvera dans un autre processus que le restant des composants.

Quand le système doit décider quel processus il doit tuer, pour libérer de la mémoire principalement, il mesure quelle est l'importance relative de chaque processus pour l'utilisateur. Par exemple, il sera plus enclin à fermer un processus qui ne contient aucune activité visible pour l'utilisateur, alors que d'autres ont des composants qui fonctionnent encore — une activité visible ou un receiver qui gère un évènement. On dit qu'une activité visible a une plus grande priorité qu'une activité non visible.

18.1.2. Threads

Quand une activité est lancée, le système crée un thread principal dans lequel s'exécutera l'application. C'est ce thread qui est en charge d'écouter les évènements déclenchés par l'utilisateur quand il interagit avec l'interface graphique. C'est pourquoi le second nom du thread principal est **thread UI** (UI pour *User Interface*, « interface utilisateur » en français).

Mais il est possible d'avoir plusieurs threads. Android utilise un *pool* de threads (comprendre une réserve de threads, pas une piscine de threads) pour gérer le multitâche. Un pool de threads comprend un certain nombre n de threads afin d'exécuter un certain nombre m de tâches (n et m n'étant pas forcément identiques) qui se trouvent dans un autre pool en attendant qu'un thread s'occupe d'elles. Logiquement, un pool est organisé comme une file, ce qui signifie qu'on empile les éléments les uns sur les autres et que nous n'avons accès qu'au sommet de cet empilement. Les résultats de chaque thread sont aussi placés dans un pool de manière à pouvoir les récupérer dans un ordre cohérent. Dès qu'un thread complète sa tâche, il va demander la prochaine tâche qui se trouve dans le pool jusqu'à ce qu'il n'y ait plus de tâches.

Avant de continuer, laissez-moi vous expliquer le fonctionnement interne de l'interface graphique. Dès que vous effectuez une modification sur une vue, que ce soit un widget ou un layout, cette modification ne se fait pas instantanément. À la place, un évènement est créé. Il génère un message, qui sera envoyé dans une pile de messages. L'objectif du thread UI est d'accéder à la pile des messages, de dépiler le premier message à traiter, de le traiter, puis de passer au suivant. De plus, ce thread s'occupe de toutes les méthodes de *callback* du système, par exemple `onCreate()` ou `onKeyDown()`. Si le système est occupé à un travail intensif, il ne pourra pas traiter les méthodes de *callback* ou les interactions avec l'utilisateur. De ce fait, un ARN est déclenché pour signaler à l'utilisateur qu'Android n'est pas d'accord avec ce comportement.

De la sorte, il faut respecter deux règles dès qu'on manipule des threads :

1. Ne jamais bloquer le thread UI.
2. Ne pas manipuler les vues standards en dehors du thread UI.

Enfin, on évite certaines opérations dans le thread UI, en particulier :

— Accès à un réseau, même s'il s'agit d'une courte opération en théorie.
— Certaines opérations dans la base de données, surtout les sélections multiples.
— Les accès fichiers, qui sont des opérations plutôt lentes.
— Enfin, les accès matériels, car certains demandent des temps de chargement vraiment trop longs.

Mais voyons un peu les techniques qui nous permettrons de faire tranquillement ces opérations.

18.2. Gérer correctement les threads simples

18.2.1. La base

En Java, un thread est représenté par un objet de type `Thread`, mais avant cela laissez-moi vous parler de l'interface `Runnable`. Cette interface représente les objets qui sont capables de faire

exécuter du code au processeur. Elle ne possède qu'une méthode, `void run()`, dans laquelle il faut écrire le code à exécuter.

Ainsi, il existe deux façons d'utiliser les threads. Comme `Thread` implémente `Runnable`, alors vous pouvez très bien créer une classe qui dérive de `Thread` afin de redéfinir la méthode `void run()`. Par exemple, ce thread fait en sorte de chercher un texte dans un livre pour le mettre dans un `TextView` :

```

```

Puis on ajoute le `Thread` à l'endroit désiré et on le lance avec `synchronized void start()` :

```

```

Une méthode `synchronized` a un verrou. Dès qu'on lance cette méthode, alors le verrou s'enclenche et il est impossible pour d'autres threads de lancer la même méthode.

Mais ce n'est pas la méthode à privilégier, car elle est contraignante à entretenir. À la place, je vous conseille de passer une instance anonyme de `Runnable` dans un `Thread` :

```

```

Le problème de notre exemple, c'est que l'opération coûteuse (la recherche d'un texte dans un livre) s'exécute dans un autre thread. C'est une bonne chose, c'est ce qu'on avait demandé, comme ça la recherche se fait sans bloquer le thread UI, mais on remarquera que la vue est aussi manipulée dans un autre thread, ce qui déroge à la seconde règle vue précédemment, qui précise que les vues doivent être manipulées dans le thread UI ! On risque de rencontrer des comportements inattendus ou impossibles à prédire !

Afin de remédier à ce problème, Android offre plusieurs manières d'accéder au thread UI depuis d'autres threads. Par exemple :

— La méthode d'`Activity` `void runOnUiThread(Runnable action)` spécifie qu'une action doit s'exécuter dans le thread UI. Si le thread actuel est le thread UI, alors l'action est exécutée immédiatement. Sinon, l'action est ajoutée à la pile des évènements du thread UI.
— Sur un `View`, on peut faire `boolean post(Runnable action)` pour ajouter le `Runnable` à la pile des messages du thread UI. Le `boolean` retourné vaut `true` s'il a été correctement placé dans la pile des messages.
— De manière presque similaire, `boolean postDelayed(Runnable action, long delayMillis)` permet d'ajouter un `Runnable` à la pile des messages, mais uniquement après qu'une certaine durée `delayMillis` s'est écoulée.

On peut par exemple voir :

```

```

Ou :

```

```

Ainsi, la longue opération s'exécute dans un thread différent, ce qui est bien, et la vue est manipulée dans le thread UI, ce qui est parfait !

Le problème ici est que ce code peut vite devenir difficile à maintenir. Vous avez vu, pour à peine deux lignes de code à exécuter, on a dix lignes d'enrobage ! Je vais donc vous présenter une solution qui permet un contrôle total tout en étant plus évidente à manipuler.

18.2.2. Les messages et les handlers

La classe `Thread` est une classe de bas niveau et en Java on préfère travailler avec des objets d'un plus haut niveau. Une autre manière d'utiliser les threads est d'utiliser la classe `Handler`, qui est d'un plus haut niveau.

En informatique, « de haut niveau » signifie « qui s'éloigne des contraintes de la machine pour faciliter sa manipulation pour les humains ».

La classe `Handler` contient un mécanisme qui lui permet d'ajouter des messages ou des `Runnable` à une file de messages. Quand vous créez un `Handler`, il sera lié à un thread, c'est donc dans la file de ce thread-là qu'il pourra ajouter des messages. Le thread UI possède lui aussi un handler et chaque handler que vous créerez communiquera par défaut avec ce handler-là.

Les handlers tels que je vais les présenter doivent être utilisés pour effectuer des calculs avant de mettre à jour l'interface graphique, et c'est tout. Ils peuvent être utilisés pour effectuer des calculs et ne pas mettre à jour l'interface graphique après, mais ce n'est pas le comportement attendu.

Mais voyons tout d'abord comment les handlers font pour se transmettre des messages. Ces messages sont représentés par la classe `Message`. Un message peut contenir un `Bundle` avec la méthode `void setData(Bundle data)`. Mais comme vous le savez, un Bundle, c'est lourd, il est alors possible de mettre des objets dans des attributs publics :

— `arg1` et `arg2` peuvent contenir des entiers.
— On peut aussi mettre un `Object` dans `obj`.

Bien que le constructeur de `Message` soit public, la meilleure manière d'en construire un est d'appeler la méthode statique `Message Message.obtain()` ou encore `Message Handler.obtainMessage()`. Ainsi, au lieu d'allouer de nouveaux objets, on récupère des anciens objets qui se trouvent dans le pool de messages. Notez que si vous utilisez la seconde méthode, le

handler sera déjà associé au message, mais vous pouvez très bien le mettre *a posteriori* avec `void setTarget(Handler target)`.

Enfin, les méthodes pour planifier des messages sont les suivantes :

- `boolean post(Runnable r)` pour ajouter `r` à la queue des messages. Il s'exécutera sur le `Thread` auquel est rattaché le `Handler`. La méthode renvoie `true` si l'objet a bien été rajouté. De manière alternative, `boolean postAtTime(Runnable r, long uptimeMillis)` permet de lancer un `Runnable` au moment `longMillis` et `boolean postDelayed(Runnable r, long delayMillis)` permet d'ajouter un `Runnable` à lancer après un délai de `delayMillis`.
- `boolean sendEmptyMessage(int what)` permet d'envoyer un `Message` simple qui ne contient que la valeur `what`, qu'on peut utiliser comme un identifiant. On trouve aussi les méthodes `boolean sendEmptyMessageAtTime(int what, long uptimeMillis)` et `boolean sendEmptyMessageDelayed(int what, long delayMillis)`.
- Pour pousser un `Message` complet à la fin de la file des messages, utilisez `boolean sendMessage(Message msg)`. On trouve aussi `boolean sendMessageAtTime(Message msg, long uptimeMillis)` et `boolean sendMessageDelayed(Message msg, long delayMillis)`.

Tous les messages seront reçus dans la méthode de *callback* `void handleMessage(Message msg)` dans le thread auquel est attaché ce `Handler`.

18.2.3. Application : une barre de progression

18.2.3.1. Énoncé

Une utilisation typique des handlers est de les incorporer dans la gestion des barres de progression. On va faire une petite application qui ne possède au départ qu'un bouton. Cliquer dessus lance un téléchargement et une boîte de dialogue s'ouvrira. Cette boîte contiendra une barre de progression qui affichera l'avancement du téléchargement, comme à la figure suivante. Dès que le téléchargement se termine, la boîte de dialogue se ferme et un `Toast` indique que le téléchargement est terminé. Enfin, si l'utilisateur s'impatiente, il peut très bien fermer la boîte de dialogue avec le bouton `Retour`.

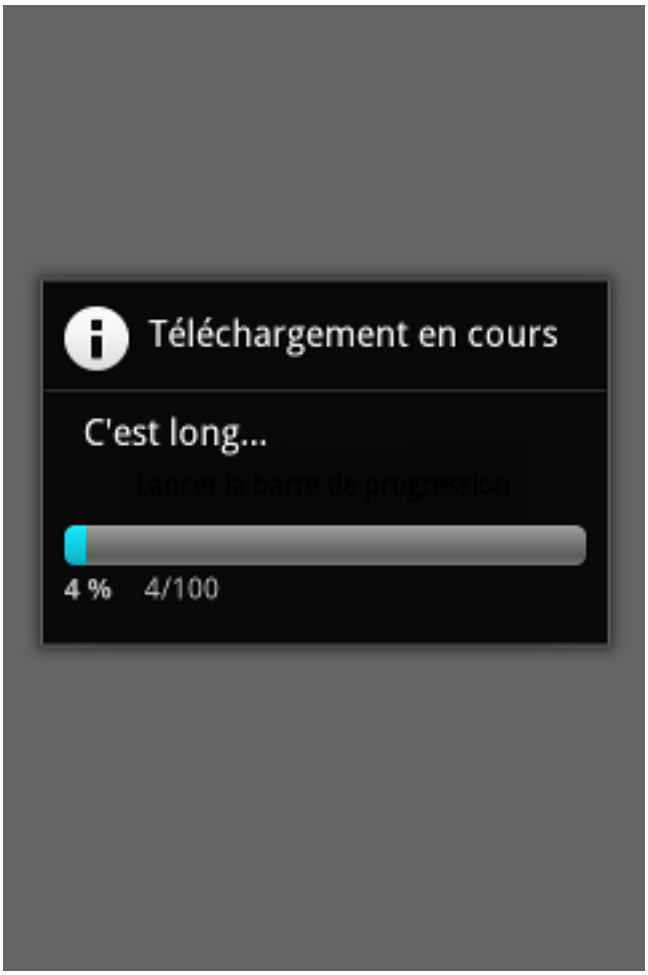

FIGURE 18.2. – Une barre de progression

18.2.3.2. Spécifications techniques

On va utiliser un `ProgressDialog` pour afficher la barre de progression. Il s'utilise comme n'importe quelle boîte de dialogue, sauf qu'il faut lui attribuer un style si on veut qu'il affiche une barre de progression. L'attribution se fait avec la méthode `setProgressStyle(int style)` en lui passant le paramètre `ProgressDialog.STYLE_HORIZONTAL`.

L'état d'avancement sera conservé dans un attribut. Comme on ne sait pas faire de téléchargement, l'avancement se fera au travers d'une boucle qui augmentera cet attribut. Bien entendu, on ne fera pas cette boucle dans le thread principal, sinon l'interface graphique sera complètement bloquée ! Alors on lancera un nouveau thread. On passera par un handler pour véhiculer des messages. On répartit donc les rôles ainsi :

— Dans le nouveau thread, on calcule l'état d'avancement, puis on l'envoie au handler à l'aide d'un message.
— Dans le handler, dès qu'on reçoit le message, on met à jour la progression de la barre.

Entre chaque incrémentation de l'avancement, allouez-vous une seconde de répit, sinon votre téléphone va faire la tête. On peut le faire avec :

```

```

Enfin, on peut interrompre un `Thread` avec la méthode `void interrupt()`. Cependant, si votre thread est en train de dormir à cause de la méthode `sleep`, alors l'interruption `InterruptedException` sera lancée et le thread ne s'interrompra pas. À vous de réfléchir pour contourner ce problème.

18.2.3.3. Ma solution

```

```

18.2.4. Sécuriser ses threads

Les threads ne sont pas des choses aisées à manipuler. À partir de notre application précédente, nous allons voir certaines techniques qui vous permettront de gérer les éventuels débordements imputés aux threads.

18.2.4.1. Il y a une fuite

Une erreur que nous avons commise est d'utiliser le handler en classe interne. Le problème de cette démarche est que quand on déclare une classe interne, alors chaque instance de cette classe contient une référence à la classe externe. Par conséquent, tant qu'il y a des messages sur la pile des messages qui sont liés au handler, l'activité ne pourra pas être nettoyée par le système, et une activité, ça pèse lourd pour le système !

Une solution simple est de faire une classe externe qui dérive de `Handler`, et de rajouter une instance de cette classe en tant qu'attribut.

```

```

Ne pas oublier d'inclure la boîte de dialogue dans le message puisque nous ne sommes plus dans la même classe ! Si vous vouliez vraiment rester dans la même classe, alors vous auriez pu déclarer `ProgressHandler` comme statique de manière à séparer les deux classes.

18.2.4.2. Gérer le cycle de l'activité

Il faut lier les threads au cycle des activités. On pourrait se dire qu'on veut parfois effectuer des tâches d'arrière-plan même quand l'activité est terminée, mais dans ce cas-là on ne passera pas par des threads mais par des `Services`, qui seront étudiés dans le prochain chapitre.

Le plus important est de gérer le changement de configuration. Pour cela, tout se fait dans `onRetainNonConfigurationInstance()`. On fait en sorte de sauvegarder le thread ainsi que la boîte de dialogue de manière à pouvoir les récupérer :

Enfin, vous pouvez aussi faire en sorte d'arrêter le thread dès que l'activité passe en pause ou quitte son cycle.

18.3. AsyncTask

Il faut avouer que tout cela est bien compliqué et nécessite de penser à tout, ce qui est source de confusion. Je vais donc vous présenter une alternative plus évidente à maîtriser, mais qui est encore une fois réservée à l'interaction avec le thread UI. `AsyncTask` vous permet de faire proprement et facilement des opérations en parallèle du thread UI. Cette classe permet d'effectuer des opérations d'arrière-plan et de publier les résultats dans le thread UI sans avoir à manipuler de threads ou de handlers.

> `AsyncTask` n'est pas une alternative radicale à la manipulation des threads, juste un substitut qui permet le travail en arrière-plan sans toucher les blocs de bas niveau comme les threads. On va surtout les utiliser pour les opérations courtes (quelques secondes tout au plus) dont on connaît précisément l'heure de départ et de fin.

On ne va pas utiliser directement `AsyncTask`, mais plutôt créer une classe qui en dérivera. Cependant, il ne s'agit pas d'un héritage évident puisqu'il faut préciser trois paramètres :

- Le paramètre `Params` permet de définir le typage des objets sur lesquels on va faire une opération.
- Le deuxième paramètre, `Progress`, indique le typage des objets qui indiqueront l'avancement de l'opération.
- Enfin, `Result` est utilisé pour symboliser le résultat de l'opération.

Ce qui donne dans le contexte :

Ensuite, pour lancer un objet de type `MaClasse`, il suffit d'utiliser dessus la méthode `final AsyncTask<Params, Progress, Result> execute (Params... params)` sur laquelle il est possible de faire plusieurs remarques :

- Son prototype est accompagné du mot-clé `final`, ce qui signifie que la méthode ne peut être redéfinie dans les classes dérivées d'`AsyncTask`.
- Elle prend un paramètre de type `Params...` ce qui pourra en troubler plus d'un, j'imagine. Déjà, `Params` est tout simplement le type que nous avons défini auparavant dans la déclaration de `MaClasse`. Ensuite, les trois points signifient qu'il s'agit d'*arguments variables* et que par conséquent on peut en mettre autant qu'on veut. Si on prend l'exemple de la méthode `int somme(int... nombres)`, on peut l'appeler avec `somme(1)` ou `somme(1,5,-2)`. Pour être précis, il s'agit en fait d'un tableau déguisé, vous pouvez donc considérer `nombres` comme un `int[]`.

Une fois cette méthode exécutée, notre classe va lancer quatre méthodes de *callback*, dans cet ordre :

— `void onPreExecute()` est lancée dès le début de l'exécution, avant même que le travail commence. On l'utilise donc pour initialiser les différents éléments qui doivent être initialisés.
— Ensuite, on trouve `Result doInBackground(Params... params)`, c'est dans cette méthode que doit être effectué le travail d'arrière-plan. À la fin, on renvoie le résultat de l'opération et ce résultat sera transmis à la méthode suivante — on utilise souvent un `boolean` pour signaler la réussite ou l'échec de l'opération. Si vous voulez publier une progression pendant l'exécution de cette méthode, vous pouvez le faire en appelant `final void publishProgress(Progress... values)` (la méthode de *callback* associée étant `void onProgressUpdate(Progress... values)`).
— Enfin, `void onPostExecute(Result result)` permet de conclure l'utilisation de l'`AsyncTask` en fonction du résultat `result` passé en paramètre.

De plus, il est possible d'annuler l'action d'un `AsyncTask` avec `final boolean cancel(boolean mayInterruptIfRunning)`, où `mayInterruptIfRunning` vaut `true` si vous autorisez l'exécution à s'interrompre. Par la suite, une méthode de *callback* est appelée pour que vous puissez réagir à cet évènement : `void onCancelled()`.

Enfin, dernière chose à savoir, un `AsyncTask` n'est disponible que pour une unique utilisation, s'il s'arrête ou si l'utilisateur l'annule, alors il faut en recréer un nouveau.

?

Et cette fois on fait comment pour gérer les changements de configuration ?

Ah ! vous aimez avoir mal, j'aime ça. Accrochez-vous parce que ce n'est pas simple. Ce que nous allons voir est assez avancé et de bas niveau, alors essayez de bien comprendre pour ne pas faire de boulettes quand vous l'utiliserez par la suite.

On pourrait garder l'activité qui a lancé l'`AsyncTask` en paramètre, mais de manière générale il ne faut jamais garder de référence à une classe qui dérive de `Context`, par exemple `Activity`. Le problème, c'est qu'on est bien obligés par moment. Alors comment faire ?

Revenons aux bases de la programmation. Quand on crée un objet, on réserve dans la mémoire allouée par le processus un emplacement qui aura la place nécessaire pour mettre l'objet. Pour accéder à l'objet, on utilise une **référence** sous forme d'un identifiant déclaré dans le code :

```
```

Ici, `chaine` est l'identifiant, autrement dit une référence qui pointe vers l'emplacement mémoire réservé pour cette chaîne de caractères.

Bien sûr, au fur et à mesure que le programme s'exécute, on va allouer de la place pour d'autres objets et, si on ne fait rien, la mémoire va être saturée. Afin de faire en sorte de libérer de la mémoire, un processus qui s'appelle le *garbage collector* (« ramasse-miettes » en français) va détruire les objets qui ne sont plus susceptibles d'être utilisés :

```

```

La variable `chaine` sera disponible avant, pendant et après le `if` puisqu'elle a été déclarée avant (donc de 1 à 5, voire plus loin encore), en revanche `dix` a été déclaré dans le `if`, il ne sera donc disponible que dedans (donc de 4 à 5). Dès qu'on sort du `if`, le *garbage collector* passe et désalloue la place réservée `dix` de manière à pouvoir l'allouer à un autre objet.

Quand on crée un objet en Java, il s'agit toujours d'une **référence forte**, c'est-à-dire que l'objet est protégé contre le *garbage collector* tant qu'on est certain que vous l'utilisez encore. Ainsi, si on garde notre activité en référence forte en tant qu'attribut de classe, elle restera toujours disponible, et vous avez bien compris que ce n'était pas une bonne idée, surtout qu'une référence à une activité est bien lourde.

À l'opposé des références fortes se trouvent les **références faibles**. Les références faibles ne protègent pas une référence du *garbage collector*.

Ainsi, si vous avez une référence forte vers un objet, le *garbage collector* ne passera pas dessus. Si vous en avez deux, idem. Si vous avez deux références fortes et une référence faible, c'est la même chose, parce qu'il y a deux références fortes.

Si le *garbage collector* réalise que l'une des deux références fortes n'est plus valide, l'objet est toujours conservé en mémoire puisqu'il reste une référence forte. *En revanche*, dès que la seconde référence forte est invalidée, alors l'espace mémoire est libéré puisqu'il ne reste plus aucune référence forte, juste une petite référence faible qui ne protège pas du ramasse-miettes.

Ainsi, il suffit d'inclure une référence faible vers notre activité dans l'`AsyncTask` pour pouvoir garder une référence vers l'activité sans pour autant la protéger contre le ramasse-miettes.

Pour créer une référence faible d'un objet `T`, on utilise `WeakReference` de cette manière :

```

```

Il n'est bien entendu pas possible d'utiliser directement un `WeakReference`, comme il ne s'agit que d'une référence faible, il vous faut donc récupérer une référence forte de cet objet. Pour ce faire, il suffit d'utiliser `T get()`. Cependant, cette méthode renverra `null` si l'objet a été nettoyé par le *garbage collector*.

18.3.1. Application

18.3.1.1. Énoncé

Faites exactement comme l'application précédente, mais avec un `AsyncTask` cette fois.

18.3.1.2. Spécifications techniques

L'`AsyncTask` est utilisé en tant que classe interne statique, de manière à ne pas avoir de fuites comme expliqué dans la partie consacrée aux threads.

Comme un `AsyncTask` n'est disponible qu'une fois, on va en recréer un à chaque fois que l'utilisateur appuie sur le bouton.

Il faut lier une référence faible à votre activité à l'`AsyncTask` pour qu'à chaque fois que l'activité est détruite on reconstruise une nouvelle référence faible à l'activité dans l'`AsyncTask`. Un bon endroit pour faire cela est dans le `onRetainNonConfigurationInstance()`.

18.3.1.3. Ma solution

```

```

Pour terminer, voici une liste de quelques comportements à adopter afin d'éviter les aléas des blocages :

— Si votre application fait en arrière-plan de gros travaux bloquants pour l'interface graphique (imaginez qu'elle doive télécharger une image pour l'afficher à l'utilisateur), alors il suffit de montrer l'avancement de ce travail avec une barre de progression de manière à faire patienter l'utilisateur.
— En ce qui concerne les jeux, usez et abusez des threads pour effectuer des calculs de position, de collision, etc.
— Si votre application a besoin de faire des initialisations au démarrage et que par conséquent elle met du temps à se charger, montrez un *splash screen* avec une barre de progression pour montrer à l'utilisateur que votre application n'est pas bloquée.

— Par défaut, au lancement d'une application, le système l'attache à un nouveau processus dans lequel il sera exécuté. Ce processus contiendra tout un tas d'informations relatives à l'état courant de l'application qu'il contient et des threads qui exécutent le code.
— Vous pouvez décider de forcer l'exécution de certains composants dans un processus à part grâce à l'attribut `android:process` à rajouter dans l'un des éléments constituant le noeud `<application>` de votre manifest.
— Lorsque vous jouez avec les threads, vous ne devez jamais perdre à l'esprit deux choses :
  — Ne jamais bloquer le thread UI.
  — Ne pas manipuler les vues standards en dehors du thread UI.
— On préfèrera toujours privilégier les concepts de haut niveau pour faciliter les manipulations pour l'humain et ainsi donner un niveau d'abstraction aux contraintes machines. Pour les threads, vous pouvez donc privilégier les messages et les handlers à l'utilisation directe de la classe `Thread`.
— `AsyncTask` est un niveau d'abstraction encore supérieur aux messages et handlers. Elle permet d'effectuer des opérations en arrière-plan et de publier les résultats dans le thread UI facilement grâce aux méthodes qu'elle met à disposition des développeurs lorsque vous en créez une.

19. Les services

Nous savons désormais faire du travail en arrière-plan, mais de manière assez limitée quand même. En effet, toutes les techniques que nous avons vues étaient destinées aux opérations courtes et/ou en interaction avec l'interface graphique, or ce n'est pas le cas de toutes les opérations d'arrière-plan. C'est pourquoi nous allons voir le troisième composant qui peut faire partie d'une application : les services.

Contrairement aux threads, les services sont conçus pour être utilisés sur une longue période de temps. En effet, les threads sont des éléments sommaires qui n'ont pas de lien particulier avec le système Android, alors que les services sont des composants et sont par conséquent intégrés dans Android au même titre que les activités. Ainsi, ils vivent au même rythme que l'application. Si l'application s'arrête, le service peut réagir en conséquence, alors qu'un thread, qui n'est pas un composant d'Android, ne sera pas mis au courant que l'application a été arrêtée si vous ne lui dites pas. Il ne sera par conséquent pas capable d'avoir un comportement approprié, c'est-à-dire la plupart du temps de s'arrêter.

19.1. Qu'est-ce qu'un service ?

Tout comme les activités, les services possèdent un cycle de vie ainsi qu'un contexte qui contient des informations spécifiques sur l'application et qui constitue une interface de communication avec le restant du système. Ainsi, on peut dire que les services sont des composants très proches des activités (et beaucoup moins des receivers, qui eux ne possèdent pas de contexte). Cette configuration leur prodigue la même grande flexibilité que les activités. En revanche, à l'opposé des activités, les services ne possèdent pas d'interface graphique : c'est pourquoi on les utilise pour effectuer des travaux d'arrière-plan.

Un exemple typique est celui du lecteur de musique. Vous laissez à l'utilisateur l'opportunité de choisir une chanson à l'aide d'une interface graphique dans une activité, puis il est possible de manipuler la chanson dans une seconde activité qui nous montre un joli lecteur avec des commandes pour modifier le volume ou mettre en pause. Mais si l'utilisateur veut regarder une page web en écoutant la musique ? Comme une activité a besoin d'afficher une interface graphique, il est impossible que l'utilisateur regarde autre chose que le lecteur quand il écoute la musique. On pourrait éventuellement envisager de passer par un receiver, mais celui-ci devrait résoudre son exécution en dix secondes, ce n'est donc pas l'idéal pour un lecteur. La solution la plus évidente est bien sûr de faire jouer la musique par un service, comme ça votre client pourra utiliser une autre application sans pour autant que la musique s'interrompe. Un autre exemple est celui du lecteur d'e-mails qui va vérifier ponctuellement si vous avez reçu un nouvel e-mail.

Il existe deux types de services :

— Les plus courants sont les services *locaux* (on trouve aussi le terme *started* ou *unbound service*), où l'activité qui lance le service et le service en lui-même appartiennent à la même application.
— Il est aussi possible qu'un service soit lancé par un composant qui appartient à une autre application, auquel cas il s'agit d'un service *distant* (on trouve aussi le terme *bound service*). Dans ce cas de figure, il existe toujours une interface qui permet la communication entre le processus qui a appelé le service et le processus dans lequel s'exécute le service. Cette communication permet d'envoyer des requêtes ou récupérer des résultats par exemple. Le fait de communiquer entre plusieurs processus s'appelle l'IPC. Il peut bien sûr y avoir plusieurs clients liés à un service.
— Il est aussi possible que le service expérimente les deux statuts à la fois. Ainsi, on peut lancer un service local et lui permettre d'accepter les connexions distantes par la suite.

Vous vous en doutez peut-être, mais un service se lance par défaut dans le même processus que celui du composant qui l'a appelé. Ce qui peut sembler plus étrange et qui pourrait vous troubler, c'est que les services s'exécutent dans le thread UI. D'ailleurs, ils ne sont pas conçus pour être exécutés en dehors de ce thread, alors n'essayez pas de le délocaliser. En revanche, si les opérations que vous allez mener dans le service risquent d'affecter l'interface graphique, vous pouvez très bien lancer un thread *dans* le service. Vous voyez la différence ? Toujours lancer un service depuis le thread principal ; mais vous pouvez très bien lancer des threads dans le service.

19.2. Gérer le cycle de vie d'un service

De manière analogue aux activités, les services traversent plusieurs étapes pendant leur vie et la transition entre ces étapes est matérialisée par des méthodes de *callback*. Heureusement, le cycle des services est plus facile à maîtriser que celui des activités puisqu'il y a beaucoup moins d'étapes. La figure suivante est un schéma qui résume ce fonctionnement.

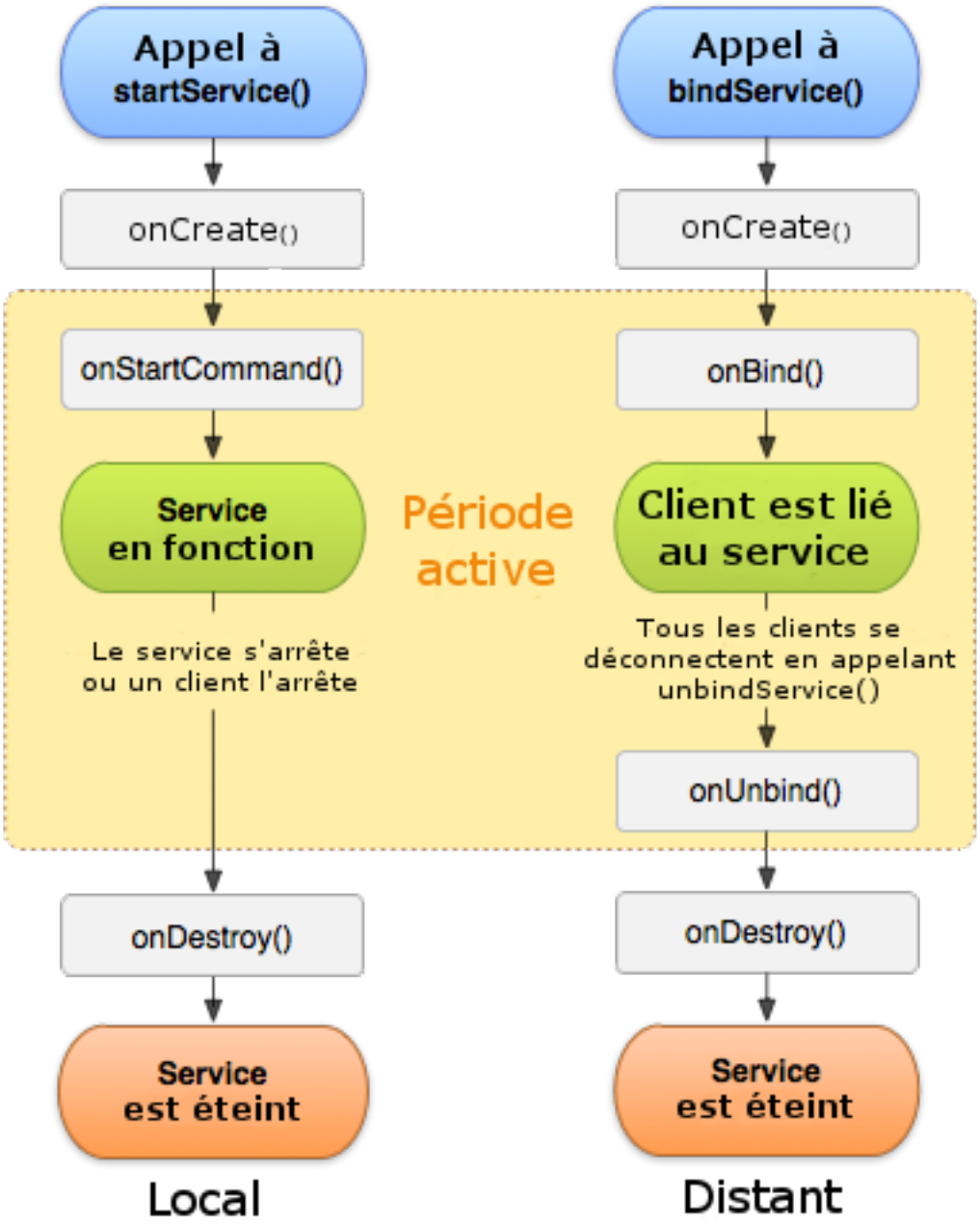

FIGURE 19.1. – Ce cycle est indépendant du cycle du composant qui a lancé le service

Vous voyez qu'on a deux cycles légèrement différents : si le service est local (lancé depuis l'application) ou distant (lancé depuis un processus différent).

19.2.1. Les services locaux

Ils sont lancés à partir d'une activité avec la méthode `ComponentName startService(Intent service)`. La variable retournée donne le package accompagné du nom du composant qui vient d'être lancé.

```
```

Si le service n'existait pas auparavant, alors il sera créé. Or, la création d'un service est symbolisée par la méthode de *callback* `void onCreate()`. La méthode qui est appelée ensuite est `int onStartCommand(Intent intent, int flags, int startId)`.

Notez par ailleurs que, si le service existait déjà au moment où vous en demandez la création avec `startService()`, alors `onStartCommande()` est appelée directement sans passer par `onCreate()`.

En ce qui concerne les paramètres, on trouve `intent`, qui a lancé le service, `flags`, dont nous discuterons juste après, et enfin `startId`, pour identifier le lancement (s'il s'agit du premier lancement du service, `startId` vaut 1, s'il s'agit du deuxième lancement, il vaut 2, etc.).

Ensuite comme vous pouvez le voir, cette méthode retourne un entier. Cet entier doit en fait être une constante qui détermine ce que fera le système s'il est tué.

19.2.1.1. START_NOT_STICKY

Si le système tue le service, alors ce dernier ne sera pas recréé. Il faudra donc effectuer un nouvel appel à `startService()` pour relancer le service.

Ce mode vaut le coup dès qu'on veut faire un travail qui peut être interrompu si le système manque de mémoire et que vous pouvez le redémarrer explicitement par la suite pour recommencer le travail. Si vous voulez par exemple mettre en ligne des statistiques sur un serveur distant. Le processus qui lancera la mise en ligne peut se dérouler toutes les 30 minutes, mais, si le service est tué avant que la mise en ligne soit effectuée, ce n'est pas grave, on le fera dans 30 minutes.

19.2.1.2. START_STICKY

Cette fois, si le système doit tuer le service, alors il sera recréé mais sans lui fournir le dernier `Intent` qui l'avait lancé. Ainsi, le paramètre `intent` vaudra `null`. Ce mode de fonctionnement est utile pour les services qui fonctionnent par intermittence, comme par exemple quand on joue de la musique.

19.2.1.3. START_REDELIVER_INTENT

Si le système tue le service, il sera recréé et dans `onStartCommand()` le paramètre `intent` sera identique au dernier intent qui a été fourni au service. `START_REDELIVER_INTENT` est indispensable si vous voulez être certains qu'un service effectuera un travail complètement.

Revenons maintenant au dernier paramètre de `onStartCommand()`, `flags`. Il nous permet en fait d'en savoir plus sur la nature de l'intent qui a lancé le service :

— 0 s'il n'y a rien de spécial à dire.

— `START_FLAG_REDELIVERY` si l'intent avait déjà été délivré et qu'il l'est à nouveau parce que le service avait été interrompu.
— Enfin vous trouverez aussi `START_FLAG_RETRY` si le service redémarre alors qu'il s'était terminé de manière anormale.

Enfin, il faut faire attention parce que `flags` n'est pas un paramètre simple à maîtriser. En effet, il peut très bien valoir `START_FLAG_REDELIVERY` et `START_FLAG_RETRY` en même temps ! Alors comment ce miracle peut-il se produire ? Laissez-moi le temps de faire une petite digression qui vous servira à chaque fois que vous aurez à manipuler des flags, aussi appelés drapeaux.

Vous savez écrire les nombres sous la forme décimale : « 0, 1, 2, 3, 4 » et ainsi de suite. On parle de numération décimale, car il y a dix unités de 0 à 9. Vous savez aussi écrire les nombres sous la forme hexadécimale : « 0, 1, 2, 3, …, 8, 9, A, B, C, D, E, F, 10, 11, 12, …, 19, 1A, 1B », et ainsi de suite. Ici, il y a seize unités de 0 à F, on parle donc d'hexadécimal. Il existe une infinité de systèmes du genre, ici nous allons nous intéresser au système binaire qui n'a que deux unités : 0 et 1. On compte donc ainsi : « 0, 1, 10, 11, 100, 101, 110, 111, 1000 », etc.

Nos trois flags précédents valent en décimal (et dans l'ordre de la liste précédente) 0, 1 et 2, ce qui fait en binaire 0, 1 et 10. Ainsi, si `flags` contient `START_FLAG_REDELIVERY` et `START_FLAG_RETRY`, alors il vaudra 1 + 2 = 3, soit en binaire 1 + 10 = 11. Vous pouvez voir qu'en fait chaque 1 correspond à la présence d'un flag : le premier à droite dénote la présence de `START_FLAG_REDELIVERY` (car `START_FLAG_REDELIVERY` vaut 1) et le plus à gauche celui de `START_FLAG_RETRY` (car `START_FLAG_RETRY` vaut 10).

On remarque tout de suite que le binaire est pratique puisqu'il permet de savoir quel flag est présent en fonction de l'absence ou non d'un 1. Mais comment demander à Java quels sont les 1 présents dans `flags` ? Il existe deux opérations de base sur les nombres binaires : le « ET » (« & ») et le « OU » (« | »). Le « ET » permet de demander « Est-ce que ce flag est présent dans `flags` ou pas ? », car il permet de vérifier que deux bits sont similaires. Imaginez, on ignore la valeur de `flags` (qui vaut « YX », on va dire) et on se demande s'il contient `START_FLAG_REDELIVERY` (qui vaut 1, soit 01 sur deux chiffres). On va alors poser l'opération comme vous le faites d'habitude :

```
```

Le résultat fait « 0X » et en fonction de X on saura si `flags` contient ou non `START_FLAG_REDELIVERY` :

— Si X vaut 0, alors `flags` ne contient pas `START_FLAG_REDELIVERY`.
— Si X vaut 1, alors il contient `START_FLAG_REDELIVERY`.

Il suffit maintenant de vérifier la valeur du résultat : s'il vaut 0, c'est que le flag n'est pas présent !

En Java, on peut le savoir de cette manière :

```
```

Je vais maintenant vous parler du « OU ». Il permet d'ajouter un flag à un nombre binaire s'il n'était pas présent auparavant :

```

```

Quelle que soit la valeur précédente de `flags`, il contient désormais `START_FLAG_RETRY`. Ainsi, si on veut vérifier qu'il ait `START_FLAG_REDELIVERY` et en même temps `START_FLAG_RETRY`, on fera :

```

```

J'espère que vous avez bien compris le concept de flags parce qu'on le retrouve souvent en programmation. Les flags permettent même d'optimiser quelque peu certains calculs pour les fous furieux, mais cela ne rentre pas dans le cadre de ce cours.

Une fois sorti de la méthode `onStartCommand()`, le service est lancé. Un service continuera à fonctionner jusqu'à ce que vous l'arrêtiez ou qu'Android le fasse de lui-même pour libérer de la mémoire RAM, comme pour les activités. Au niveau des priorités, les services sont plus susceptibles d'être détruits qu'une activité située au premier plan, mais plus prioritaires que les autres processus qui ne sont pas visibles. La priorité a néanmoins tendance à diminuer avec le temps : plus un service est lancé depuis longtemps, plus il a de risques d'être détruit. De manière générale, on va apprendre à concevoir nos services de manière à ce qu'ils puissent gérer la destruction et le redémarrage.

Pour arrêter un service, il est possible d'utiliser `void stopSelf()` depuis le service ou `boolean stopService(Intent service)` depuis une activité, auquel cas il faut fournir `service` qui décrit le service à arrêter.

Cependant, si votre implémentation du service permet de gérer une accumulation de requêtes (un pool de requêtes), vous pourriez vouloir faire en sorte de ne pas interrompre le service avant que toutes les requêtes aient été gérées, même les nouvelles. Pour éviter ce cas de figure, on peut utiliser `boolean stopSelfResult(int startId)` où `startId` correspond au même `startId` qui était fourni à `onStartCommand()`. On l'utilise de cette manière : vous lui passez un `startId` et, s'il est identique au dernier `startId` passé à `onStartCommand()`, alors le service s'interrompt. Sinon, c'est qu'il a reçu une nouvelle requête et qu'il faudra la gérer avant d'arrêter le service.

Comme pour les activités, si on fait une initialisation qui a lieu dans `onCreate()` et qui doit être détruite par la suite, alors on le fera dans le `onDestroy()`. De plus, si un service est détruit par manque de RAM, alors le système ne passera pas par la méthode `onDestroy()`.

19.2.2. Les services distants

Comme les deux types de services sont assez similaires, je ne vais présenter ici que les différences.

On utilisera cette fois `boolean bindService(Intent service, ServiceConnection conn, int flags)` afin d'assurer une connexion persistante avec le service. Le seul paramètre que vous ne connaissez pas est `conn` qui permet de recevoir le service quand celui-ci démarrera et permet de savoir s'il meurt ou s'il redémarre.

Un `ServiceConnection` est une interface pour surveiller l'exécution du service distant et il incarne le pendant client de la connexion. Il existe deux méthodes de *callback* que vous devrez redéfinir :

1. `void onServiceConnected(ComponentName name, IBinder service)` qui est appelée quand la connexion au service est établie, avec un `IBinder` qui correspond à un canal de connexion avec le service.
2. `void onServiceDisconnected(ComponentName name)` qui est appelée quand la connexion au service est perdue, en général parce que le processus qui accueille le service a planté ou a été tué.

Mais qu'est-ce qu'un `IBinder` ? Comme je l'ai déjà dit, il s'agit d'un pont entre votre service et l'activité, mais au niveau du service. Les `IBinder` permettent au client de demander des choses au service. Alors, comment créer cette interface ? Tout d'abord, il faut savoir que le `IBinder` qui sera donné à `onServiceConnected(ComponentName, IBinder)` est envoyé par la méthode de *callback* `IBinder onBind(Intent intent)` dans `Service`. Maintenant, il suffit de créer un `IBinder`. Nous allons voir la méthode la plus simple, qui consiste à permettre à l'`IBinder` de renvoyer directement le `Service` de manière à pouvoir effectuer des commandes dessus.

```
```

Le service sera créé s'il n'était pas déjà lancé (appel à `onCreate()` donc), mais ne passera pas par `onStartCommand()`.

Pour qu'un client puisse se détacher d'un service, il peut utiliser la méthode `void unbindService(ServiceConnection conn)` de `Context`, avec `conn` l'interface de connexion fournie précédemment à `bindService()`.

Ainsi, voici une implémentation typique d'un service distant :

```
```

À noter aussi que, s'il s'agit d'un service distant, alors il aura une priorité supérieure ou égale à la priorité de son client le plus important (avec la plus haute priorité). Ainsi, s'il est lié à un client qui se trouve au premier plan, il y a peu de risques qu'il soit détruit.

19.3. Créer un service

19.3.1. Dans le Manifest

Tout d'abord, il faut déclarer le service dans le Manifest. Il peut prendre quelques attributs que vous connaissez déjà tels que `android:name` qui est indispensable pour préciser son identifiant, `android:icon` pour indiquer un drawable qui jouera le rôle d'icône, `android:permission` pour créer une permission nécessaire à l'exécution du service ou encore `android:process` afin de préciser dans quel processus se lancera ce service. Encore une fois, `android:name` est le seul attribut indispensable :

```
```

De cette manière, le service se lancera dans un processus différent du reste de l'application et ne monopolisera pas le thread UI. Vous pouvez aussi déclarer des filtres d'intents pour savoir quels intents implicites peuvent démarrer votre service.

19.3.2. En Java

Il existe deux classes principales depuis lesquelles vous pouvez dériver pour créer un service.

19.3.2.1. Le plus générique : Service

La classe `Service` permet de créer un service de base. Le code sera alors exécuté dans le thread principal, alors ce sera à vous de créer un nouveau thread pour ne pas engorger le thread UI.

19.3.2.2. Le plus pratique : IntentService

En revanche la classe `IntentService` va créer elle-même un thread et gérer les requêtes que vous lui enverrez dans une file. À chaque fois que vous utiliserez `startService()` pour lancer ce service, la requête sera ajoutée à la file et tous les éléments de la file seront traités par ordre d'arrivée. Le service s'arrêtera dès que la file sera vide. Usez et abusez de cette classe, parce que la plupart des services n'ont pas besoin d'exécuter toutes les requêtes en même temps, mais plutôt les unes après les autres. En plus, elle est plus facile à gérer puisque vous aurez juste à implémenter `void onHandleIntent(Intent intent)` qui recevra toutes les requêtes dans l'ordre sous la forme d'`intent`, ainsi qu'un constructeur qui fait appel au constructeur d'`IntentService` :

```
```

Vous pouvez aussi implémenter les autres méthodes de *callback*, mais faites toujours appel à leur superimplémentation, sinon votre service échouera lamentablement :

On veut un exemple, on veut un exemple !

Je vous propose de créer une activité qui va envoyer un chiffre à un `IntentService` qui va afficher la valeur de ce chiffre dans la console. De plus, l'`IntentService` fera un long traitement pour que chaque fois que l'activité envoie un chiffre les intents s'accumulent, ce qui fera que les messages seront retardés dans la console.

J'ai une activité toute simple qui se lance au démarrage de l'application :

Cliquer sur le bouton incrémente le compteur et envoie un intent qui lance un service qui s'appelle `IntentServiceExample`. L'intent est ensuite reçu et traité :

Allez-y maintenant, cliquez sur le bouton. La première fois, le chiffre s'affichera immédiatement dans la console, mais si vous continuez vous verrez que le compteur augmente, et pas l'affichage, tout simplement parce que le traitement prend du temps et que l'affichage est retardé entre chaque pression du bouton. Cependant, chaque intent est traité, dans l'ordre d'envoi.

19.4. Les notifications et services de premier plan

19.4.1. Distribuer des autorisations

Les `PendingIntents` sont des `Intents` avec un objectif un peu particulier. Vous les créez dans votre application, et ils sont destinés à une autre application, jusque là rien de très neuf sous le soleil ! Cependant, en donnant à une autre application un `PendingIntent`, vous lui donnez les droits d'effectuer une opération comme s'il s'agissait de votre application (avec les mêmes permissions et la même identité).

En d'autres termes, vous avez deux applications : celle de départ, celle d'arrivée. Vous donnez à l'application d'arrivée tous les renseignements et toutes les autorisations nécessaires pour qu'elle puisse demander à l'application de départ d'exécuter une action à sa place.

Comment peut-on indiquer une action à effectuer ?

Vous connaissez déjà la réponse, j'en suis sûr ! On va insérer dans le `PendingIntent`… un autre `Intent`, qui décrit l'action qui sera à entreprendre. Le seul but du `PendingIntent` est d'être

véhiculé entre les deux applications (ce n'est donc pas surprenant que cette classe implémente `Parcelable`), pas de lancer un autre composant.

Il existe trois manières d'appeler un `PendingIntent` en fonction du composant que vous souhaitez démarrer. Ainsi, on utilisera l'une des méthodes statiques suivantes :

```
```

Comme vous l'aurez remarqué, les paramètres sont toujours les mêmes :

— `context` est le contexte dans lequel le `PendingIntent` devrait démarrer le composant.
— `requestCode` est un code qui n'est pas utilisé.
— `intent` décrit le composant à lancer (dans le cas d'une activité ou d'un service) ou l'`Intent` qui sera diffusé (dans le cas d'un `broadcast`).
— `flags` est également assez peu utilisé.

Le `PendingIntent` sera ensuite délivré au composant destinataire comme n'importe quel autre `Intent` qui aurait été appelé avec `startActivityForResult()` : le résultat sera donc accessible dans la méthode de *callback* `onActivityResult()`.

Voici un exemple qui montre un `PendingIntent` qui sera utilisé pour revenir vers l'activité principale :

```
```

19.4.2. Notifications

Une fois lancé, un service peut avertir l'utilisateur des évènements avec les `Toasts` ou des notifications dans la barre de statut, comme à la figure suivante.

FIGURE 19.2. – Ma barre de statut contient déjà deux notifications représentées par deux icônes à gauche

Comme vous connaissez les `Toasts` mieux que certaines personnes chez Google, je ne vais parler que des notifications.

Une notification n'est pas qu'une icône dans la barre de statut, en fait elle traverse trois étapes :

1. Tout d'abord, à son arrivée, elle affiche une icône ainsi qu'un texte court que Google appelle bizarement un « texte de téléscripteur ».
2. Ensuite, seule l'icône est lisible dans la barre de statut après quelques secondes.
3. Puis il est possible d'avoir plus de détails sur la notification en ouvrant la liste des notifications, auquel cas on peut voir une icône, un titre, un texte et un horaire de réception.

Si l'utilisateur déploie la liste des notifications et appuie sur l'une d'elles, Android actionnera un `PendingIntent` qui est contenu dans la notification et qui sera utilisé pour lancer un composant (souvent une activité, puisque l'utilisateur s'attendra à pouvoir effectuer quelque chose). Vous pouvez aussi configurer la notification pour qu'elle s'accompagne d'un son, d'une vibration ou d'un clignotement de la LED.

Les notifications sont des instances de la classe `Notification`. Cette classe permet de définir les propriétés de la notification, comme l'icône, le message associé, le son à jouer, les vibrations à effectuer, etc.

Il existe un constructeur qui permet d'ajouter les éléments de base à une notification : `Notification(int icon, CharSequence tickerText, long when)` où `icon` est une référence à un `Drawable` qui sera utilisé comme icône, `tickerText` est le texte de type téléscripteur qui sera affiché dans la barre de statut, alors que `when` permet d'indiquer la date et l'heure qui accompagneront la notification. Par exemple, pour une notification lancée dès qu'on appuie sur un bouton, on pourrait avoir :

```
```

La figure suivante représente la barre de statut avant la notification.

FIGURE 19.3. – Avant la notification

La figure suivante représente la barre de statut au moment où l'on reçoit la notification.

FIGURE 19.4. – Au moment de la notification

19.4.2.1. Ajouter du contenu à une notification

Une notification n'est pas qu'une icône et un léger texte dans la barre de statut, il est possible d'avoir plus d'informations quand on l'affiche dans son intégralité et elle *doit* afficher du contenu, au minimum un titre et un texte, comme à la figure suivante.

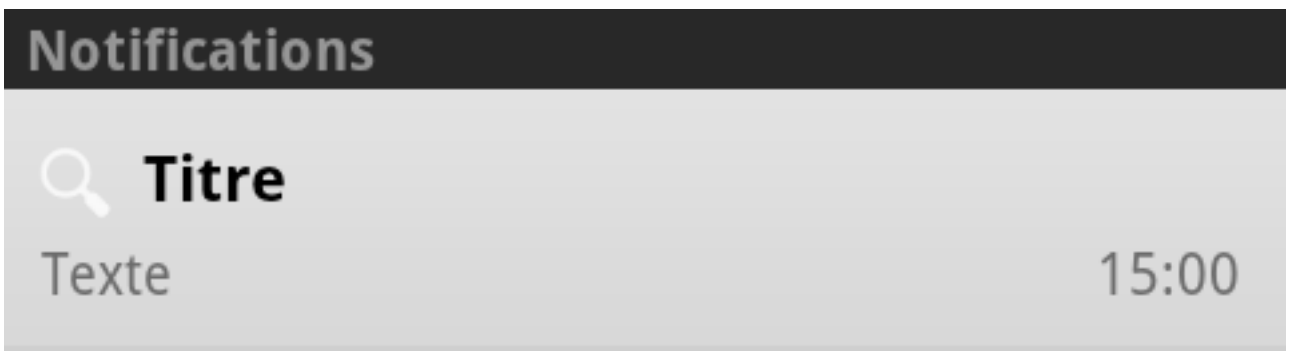

FIGURE 19.5. – La notification contient au moins un titre et un texte

De plus, il faut définir ce qui va se produire dès que l'utilisateur cliquera sur la notification. Nous allons rajouter un `PendingIntent` à la notification, et dès que l'utilisateur cliquera sur la notification, l'intent à l'intérieur de la notification sera déclenché.

Notez bien que, si l'intent lance une activité, alors il faut lui rajouter le flag `FLAG_ACTIVITY_NEW_TASK`. Ces trois composants, titre, texte et `PendingIntent` sont à définir avec la méthode `void setLatestEventInfo(Context context, CharSequence contentTitle, CharSequence contentText, PendingIntent contentIntent)`, où `contentTitle` sera le titre affiché et `contentText`, le texte. Par exemple, pour une notification qui fait retourner dans la même activité que celle qui a lancé la notification :

```
```

Enfin, il est possible de rajouter des flags à une notification afin de modifier son comportement :

— `FLAG_AUTO_CANCEL` pour que la notification disparaisse dès que l'utilisateur appuie dessus.
— `FLAG_ONGOING_EVENT` pour que la notification soit rangée sous la catégorie « En cours » dans l'écran des notifications, comme à la figure suivante. Ainsi, l'utilisateur saura que le composant qui a affiché cette notification est en train de faire une opération.

FIGURE 19.6. – La notification est rangée sous la catégorie « En cours » dans l'écran des notifications

Les flags s'ajoutent à l'aide de l'attribut `flags` qu'on trouve dans chaque notification :

```
```

19.4.2.2. Gérer vos notifications

Votre application n'est pas la seule à envoyer des notifications, toutes les applications peuvent le faire ! Ainsi, pour gérer toutes les notifications de toutes les applications, Android fait appel à un gestionnaire de notifications, représenté par la classe `NotificationManager`. Comme il n'y a qu'un `NotificationManager` pour tout le système, on ne va pas en construire un nouveau, on va plutôt récupérer celui du système avec une méthode qui appartient à la classe `Context` : `Object getSystemService(Context.NOTIFICATION_SERVICE)`. Alors réfléchissons : cette méthode appartient à `Context`, pouvez-vous en déduire quels sont les composants qui peuvent invoquer le `NotificationManager` ? Eh bien, les `Broadcast Receiver` n'ont pas de contexte, alors ce n'est pas possible. En revanche, les activités et les services peuvent le faire !

Il est ensuite possible d'envoyer une notification avec la méthode `void notify(int id, Notification notification)` où `id` sera un identifiant unique pour la notification et où on devra insérer la `notification`.

Ainsi, voici le code complet de notre application qui envoie une notification pour que l'utilisateur puisse la relancer en cliquant sur une notification :

```
```

19.4.3. Les services de premier plan

Pourquoi avons-nous appris tout cela ? Cela n'a pas grand-chose à voir avec les services ! En fait, tout ce que nous avons appris pourra être utilisé pour manipuler des services de premier plan.

? Mais cela n'a pas de sens, pourquoi voudrait-on que nos services soient au premier plan ?

Et pourquoi pas ? En fait, parler d'un service de premier plan est un abus de langage, parce que ce type de services reste un service, il n'a pas d'interface graphique, en revanche il a la même priorité qu'une activité consultée par un utilisateur, c'est-à-dire la priorité maximale. Il est donc peu probable que le système le ferme.

Il faut cependant être prudent quand on les utilise. En effet, ils ne sont pas destinés à tous les usages. On ne fait appel aux services de premier plan que si l'utilisateur sait pertinemment qu'il y a un travail en cours qu'il ne peut pas visualiser, tout en lui laissant des contrôles sur ce travail pour qu'il puisse intervenir de manière permanente. C'est pourquoi on utilise une notification qui sera une passerelle entre votre service et l'utilisateur. Cette notification devra permettre à l'utilisateur d'ouvrir des contrôles dans une activité pour arrêter le service.

Par exemple, un lecteur multimédia qui joue de la musique depuis un service devrait s'exécuter sur le premier plan, de façon à ce que l'utilisateur soit conscient de son exécution. La notification pourrait afficher le titre de la chanson, son avancement et permettre à l'utilisateur d'accéder aux contrôles dans une activité.

Pour faire en sorte qu'un service se lance au premier plan, on appelle `void startForeground(int id, Notification notification)`. Comme vous pouvez le voir, vous devez fournir un identifiant pour la notification avec `id`, ainsi que la `notification` à afficher.

```
```

Vous pouvez ensuite enlever le service du premier plan avec `void stopForeground(boolean removeNotification)`, ou vous pouvez préciser si vous voulez que la notification soit supprimée avec `removeNotification` (sinon le service sera arrêté, mais la notification persistera). Vous pouvez aussi arrêter le service avec les méthodes traditionnelles, auquel cas la notification sera aussi supprimée.

19.5. Pour aller plus loin : les alarmes

Il arrive parfois qu'on ait besoin de lancer des travaux à intervalles réguliers. C'est même indispensable pour certaines opérations : vérifier les e-mails de l'utilisateur, programmer une sonnerie tous les jours à la même heure, etc. Avec notre savoir, il existe déjà des solutions, mais rien qui permette de le faire de manière élégante !

La meilleure manière de faire est d'utiliser les alarmes. Une alarme est utilisée pour déclencher un `Intent` à intervalles réguliers.

Encore une fois, toutes les applications peuvent envoyer des alarmes, Android a donc besoin d'un système pour gérer toutes les alarmes, les envoyer au bon moment, etc. Ce système s'appelle `AlarmManager` et il est possible de le récupérer avec `Object context.getSystemService(Context.ALARM_SERVICE)`, un peu comme pour `NotificationManager`.

Il existe deux types d'alarme : les uniques et celles qui se répètent.

19.5.1. Les alarmes uniques

Pour qu'une alarme ne se déclenche qu'une fois, on utilise la méthode `void set(int type, long triggerAtMillis, PendingIntent operation)` sur l'`AlarmManager`.

On va commencer par le paramètre `triggerAtMillis`, qui définit à quel moment l'alarme se lancera. Le temps doit y être exprimé en millisecondes comme d'habitude, alors on utilisera la classe `Calendar`, que nous avons vue précédemment.

Ensuite, le paramètre `type` permet de définir le comportement de l'alarme vis à vis du paramètre `triggerAtMillis`. Est-ce que `triggerAtMillis` va déterminer le moment où l'alarme doit se déclencher (le 30 mars à 08 :52) ou dans combien de temps elle doit se déclencher (dans 25 minutes et 55 secondes) ? Pour définir une date exacte on utilisera la constante `RTC`, sinon pour un compte à rebours on utilisera `ELAPSED_REALTIME`. De plus, est-ce que vous souhaitez que l'alarme réveille l'appareil ou qu'elle se déclenche d'elle-même quand l'appareil sera réveillé d'une autre manière ? Si vous souhaitez que l'alarme réveille l'appareil rajoutez _WAKEUP aux constantes que nous venons de voir. On obtient ainsi `RTC_WAKEUP` et `ELAPSED_REALTIME_WAKEUP`.

Enfin, `operation` est le `PendingIntent` qui contient l'`Intent` qui sera enclenché dès que l'alarme se lancera.

Ainsi, pour une alarme qui se lance maintenant, on fera :

```
```

Pour une alarme qui se lancera pour mon anniversaire (notez-le dans vos agendas !), tout en réveillant l'appareil :

```
```

Et pour une alarme qui se lance dans 20 minutes et 50 secondes :

```
```

19.5.2. Les alarmes récurrentes

Il existe deux méthodes pour définir une alarme récurrente. La première est `void setRepeating(int type, long triggerAtMillis, long intervalMillis, PendingIntent operation)` qui prend les mêmes paramètres que précédemment à l'exception de `intervalMillis` qui est l'intervalle entre deux alarmes. Vous pouvez écrire n'importe quelle durée, cependant il existe quelques constantes qui peuvent vous aider :

- `INTERVAL_FIFTEEN_MINUTES` représente un quart d'heure.
- `INTERVAL_HALF_HOUR` représente une demi-heure.
- `INTERVAL_HOUR` représente une heure.
- `INTERVAL_HALF_DAY` représente 12 heures.
- `INTERVAL_DAY` représente 24 heures.

Vous pouvez bien entendu faire des opérations, par exemple `INTERVAL_HALF_DAY = INTERVAL_DAY / 2`. Pour obtenir une semaine, on peut faire `INTERVAL_DAY * 7`.

Si une alarme est retardée (parce que l'appareil est en veille et que le mode choisi ne réveille pas l'appareil par exemple), une requête manquée sera distribuée dès que possible. Par la suite, les alarmes seront à nouveau distribuées en fonction du plan originel.

Le problème de cette méthode est qu'elle est assez peu respectueuse de la batterie, alors si le délai de répétition est inférieur à une heure, on utilisera plutôt `void setInexactRepeating(int type, long triggerAtMillis, long intervalMillis, PendingIntent operation)`, auquel cas l'alarme n'est pas déclenchée au moment précis si c'est impossible.

Une alarme ne persiste pas après un redémarrage du périphérique. Si vous souhaitez que vos alarmes se réactivent à chaque démarrage du périphérique, il vous faudra écouter le `Broadcast Intent` appelé `ACTION_BOOT_COMPLETED`.

19.5.3. Annuler une alarme

Pour annuler une alarme, il faut utiliser la méthode `void cancel(PendingIntent operation)` où `operation` est le même `PendingIntent` qui accompagnait l'alarme. Si plusieurs alarmes utilisent le même `PendingIntent`, alors elles sont toutes annulées.

Il faut que tous les champs du `PendingIntent` soient identiques, à l'exception du champ `Données`. De plus, les deux `PendingIntent` doivent avoir le même identifiant.

— Les services sont des composants très proches des activités puisqu'ils possèdent un contexte et un cycle de vie similaire mais ne possèdent pas d'interface graphique. Ils sont donc destinés à des travaux en arrière-plan.
— Il existe deux types de services :
  — Les services locaux où l'activité qui lance le service et le service en lui-même appartiennent à la même application.
  — Les services distants où le service est lancé par l'activité d'une activité d'une autre application du système.
— Le cycle de vie du service est légèrement différent selon qu'il soit lancé de manière locale ou distante.
— La création d'un service se déclare dans le `manifest` dans un premier temps et se crée dans le code Java en étendant la classe `Service` ou `IntentService` dans un second temps.
— Bien qu'il soit possible d'envoyer des notifications à partir d'une activité, les services sont particulièrement adaptés pour les lancer à la fin du traitement pour lequel ils sont destinés, par exemple.
— Les alarmes sont utiles lorsque vous avez besoin d'exécuter du code à un intervalle régulier.

20. Le partage de contenus entre applications

L'avantage des bases de données, c'est qu'elles facilitent le stockage de données complexes et structurées. Cependant, le problème qu'on rencontre avec ces bases, c'est qu'il n'est pas possible d'accéder à la base de données d'une application qui ne nous appartient pas. Néanmoins, il peut arriver qu'on ait vraiment besoin de partager du contenu entre plusieurs applications. Un exemple simple et courant est de pouvoir consulter les contacts de l'utilisateur qui sont enregistrés dans l'application « Carnet d'adresses ». Ces accès aux données d'une application différente de la nôtre se font à l'aide des **fournisseurs de contenu** ou *content providers* en anglais.

Les fournisseurs de contenu sont le quatrième et dernier composant des applications que nous verrons. Techniquement, un fournisseur de contenu est découpé en deux éléments distincts :

- **Le fournisseur de contenu**, qui sera utilisé dans l'application qui distribue son contenu aux autres applications.
- **Un client**, qui permettra aux autres applications de demander au fournisseur les informations voulues.

Ensemble, les fournisseurs et les clients offrent une interface standardisée permettant l'échange sécurisé de données, ainsi que les communications inter-processus, de façon à faciliter les transactions entre applications. Ils permettent entre autres d'effectuer des copier/coller de données complexes depuis votre application vers d'autres applications.

Pour être tout à fait franc, il n'est pas rare qu'une application ne développe pas son propre fournisseur de contenu, car ils ne sont nécessaires que pour des besoins bien spécifiques, mais il se pourrait bien qu'un jour vous rencontriez ce type de difficultés.

i

Je reprends ici la même base de données qui représente les membres du Site du Zéro qui participent à l'écriture de ce tutoriel. N'hésitez pas à aller relire complètement le chapitre sur les bases de données afin de vous familiariser avec cette architecture et vous remémorer les différentes techniques et termes techniques, ce chapitre-là étant intimement lié au présent chapitre.

20.1. Côté client : accéder à des fournisseurs

Les fournisseurs de contenu permettent l'encapsulation de données, et, pour accéder à ces données, il faudra utiliser les fameuses URI. Ici, nous ne saurons pas où ni comment les données

sont stockées. Dans une base de données, dans un fichier, sur un serveur distant ? Cela ne nous regarde pas, du moment que les données nous sont mises à disposition.

Cependant, quel que soit le type de stockage, les données nous sont toujours présentées de la même manière. En effet, et un peu comme une base de données relationnelle, un fournisseur de contenu présente les données à une application extérieure dans une ou plusieurs tables. Chaque entrée dans la table est représentée dans une ligne et chaque colonne représente un attribut.

Une chose importante à savoir avant de faire appel à un fournisseur de contenu : vous devez savoir par avance la structure des tables (ses attributs et les valeurs qu'ils peuvent prendre), car vous en aurez besoin pour exploiter correctement ce fournisseur de contenu. Il n'y a pas moyen d'obtenir ce genre d'informations, il faut que le développeur du fournisseur vous communique cette information.

Un fournisseur n'a pas besoin d'avoir une clé primaire. S'il en a une, elle peut l'appeler `_ID`, même si ce n'est pas nécessaire. Si vous le faites, alors Android pourra faire quelques traitements automatiquement. Par exemple, si vous voulez lier des données depuis un fournisseur vers une `ListView`, il vaut mieux que la clé primaire s'appelle `_ID` de façon à ce que le `ListView` puisse deviner tout seul qu'il s'agit d'une clé primaire.

Examinons les éléments architecturaux des fournisseurs de contenu, ainsi que la relation qui existe entre les fournisseurs de contenu et les autres abstractions qui permettent l'accès aux données.

20.1.1. Accéder à un fournisseur

Il est possible d'accéder aux données d'une autre application avec un objet client `ContentResolver`. Cet objet a des méthodes qui appellent d'autres méthodes, qui ont le même nom, mais qui se trouvent dans un objet fournisseur, c'est-à-dire l'objet qui met à disposition le contenu pour les autres applications. Les objets fournisseurs sont de type `ContentProvider`. Aussi, si votre `ContentResolver` a une méthode qui s'appelle `myMethod`, alors le `ContentProvider` aura aussi une méthode qui s'appelle `myMethod`, et quand vous appelez `myMethod` sur votre `ContentResolver`, il fera en sorte d'appeler `myMethod` sur le `ContentProvider`.

Pourquoi je n'irais pas appeler ces méthodes moi-même ? Cela irait plus vite et ce serait plus simple !

Parce que ce n'est pas assez sécurisé ! Avec ce système, Android est certain que vous avez reçu les autorisations nécessaires à l'exécution de ces opérations.

Vous vous rappelez ce qu'on avait fait pour les bases de données ? On avait écrit des méthodes qui permettent de créer, lire, mettre à jour ou détruire des informations dans une base de données. Eh bien, ces méthodes, appelées méthodes CRUD, sont fournies par le `ContentResolver`. Ainsi, si mon `ContentResolver` demande poliment à un `ContentProvider` de lire des entrées dans la base de données de l'application dans laquelle se trouve ce `ContentProvider`, il appellera

sur lui-même la méthode `lireCesDonnées` pour que soit appelée sur le `ContentProvider` la même méthode `lireCesDonnées`.

L'objet de type `ContentResolver` dans le processus de l'application cliente et l'objet de type `ContentProvider` de l'application qui fournit les données gèrent automatiquement les communications inter-processus, ce qui est bien parce que ce n'est pas une tâche aisée du tout. `ContentProvider` sert aussi comme une couche d'abstraction entre le référentiel de données et l'apparence extérieure des données en tant que tables.

Pour accéder à un fournisseur, votre application a besoin de certaines permissions. Vous ne pouvez bien entendu pas utiliser n'importe quel fournisseur sans l'autorisation de son application mère ! Nous verrons comment utiliser ou créer une permission par la suite.

Pour récupérer le gestionnaire des fournisseurs de contenu, on utilise la méthode de `Context` appelée `ContentResolver getContentResolver ()`. Vous aurez ensuite besoin d'une URI pour déterminer à quel fournisseur de contenu vous souhaitez accéder.

20.1.2. L'URI des fournisseurs de contenu

Le **schéma** d'une URI qui représente un fournisseur de contenu est `content`. Ainsi, ce type d'URI commence par `content://`.

Après le schéma, on trouve l'**information**. Comme dans le cas des URL sur internet, cette information sera un chemin. Ce chemin est dit hiérarchique : plus on rajoute d'informations, plus on devient précis sur le contenu voulu. La première partie du chemin s'appelle l'**autorité**. Elle est utilisée en tant qu'identifiant unique afin de pouvoir différencier les fournisseurs dans le registre des fournisseurs que tient Android. Un peu comme un nom de domaine sur internet. Si vous voulez aller sur le Site du Zéro , vous utiliserez le nom de domaine `www.siteduzero.com`. Ici, le schéma est `http` (dans le cas d'une URL, le schéma est le protocole de communication utilisé pour recevoir et envoyer des informations) et l'autorité est `www.siteduzero.com`, car elle permet de retrouver le site de manière unique. Il n'y a aucun autre site auquel vous pourrez accéder en utilisant l'adresse `www.siteduzero.com`.

Si on veut rentrer dans une partie spécifique du Site du Zéro, on va ajouter des composantes au chemin et chaque composante permet de préciser un peu plus l'emplacement ciblé : `http:/www.siteduzero.com/forum/android/demande_d_aide.html` (cette URL est bien entendu totalement fictive).

Comme vous pouvez le voir, les composantes sont séparées par des « / ». Ces composantes sont appelées des **segments**. On retrouve ainsi le segment `forum` qui nous permet de savoir qu'on se dirige vers les forums, puis `android` qui permet de savoir qu'on va aller sur un forum dédié à Android, et enfin `demande_d_aide.html` qui permet de se diriger vers le forum Android où on peut demander de l'aide.

Les URI pour les fournisseurs de contenu sont similaires. L'autorité seule est totalement nécessaire et chaque segment permet d'affiner un peu la recherche. Par exemple, il existe une API pour accéder aux données associées aux contacts enregistrés dans le téléphone : `Contacts Contract`. Elle possède plusieurs tables, dont `ContactsContract.Data` qui contient des données sur les contacts (numéros de téléphone, adresses e-mail, comptes Facebook, etc.),

`ContactsContract.RawContacts` qui contient les contacts en eux-mêmes, et enfin `ContactsContract.Contacts` qui fait le lien entre ces deux tables, pour lier un contact à ses données personnelles.

Pour accéder à `ContactsContract`, on peut utiliser l'URI `content://com.android.contacts/`. Si je cherche uniquement à accéder à la table `Contact`, je peux utiliser l'URI `content://com.android.contacts/contact`. Néanmoins, je peux affiner encore plus la recherche en ajoutant un autre segment qui indiquera l'identifiant du contact recherché : `content://com.android.contacts/contact/18`.

Ainsi, si j'effectue une recherche avec `content://com.android.contacts/contact` sur mon téléphone, j'aurai 208 résultats, alors que si j'utilise `content://com.android.contacts/contact/18` je n'aurai qu'un résultat, celui d'identifiant 18.

De ce fait, le schéma sera `content://` et l'autorité sera composée du nom du package. Le premier segment indiquera la table dans laquelle il faut chercher et le deuxième la composante de la ligne à récupérer : `content://sdz.chapitreQuatre.Provider/Client/5`. Ici, je récupère la cinquième entrée de ma table `Client` dans mon application `Provider` qui se situe dans le package `sdz.chapitreQuatre`.

On ne pourra retrouver une ligne que si l'on a défini un identifiant en lui donnant comme nom de colonne `_ID`. Dans l'exemple précédent, on cherche dans la table `Client` celui qui a pour valeur 5 dans la colonne `_ID`.

Android possède nativement un certain nombre de fournisseurs de contenu qui sont décrits dans `android.provider`. Vous trouverez une liste de ces fournisseurs sur la documentation . On trouve parmi ces fournisseurs des accès aux données des contacts, des appels, des médias, etc. Chacune de ces classes possède une constante appelée `CONTENT_URI` qui est en fait l'URI pour accéder au fournisseur qu'elles incarnent. Ainsi, pour accéder au fournisseur de contenu de `ContactsContract.Contacts`, on pourra utiliser l'URI `ContactsContract.Contacts.CONTENT_URI`.

Vous remarquerez que l'autorité des fournisseurs de contenu d'Android ne respecte pas la tradition qui veut qu'on ait le nom du package ainsi que le nom du fournisseur. Google peut se le permettre mais pas vous, alors n'oubliez pas la bonne procédure à suivre.

On trouve par exemple :

Nom	Description	Interface
Contact	**Permet l'accès aux données des contacts de l'utilisateur.**	**La base est `ContactsContract`, mais il existe une vingtaine de façons d'accéder à ces informations.**

Magasin multimédia	Liste les différents médias disponibles sur le support, tels que les images, vidéos, fichiers audios, etc.	La base est `MediaStore`, mais il existe encore une fois un bon nombre de dérivés, par exemple `MediaStore.Audio.Artists` liste tous les artistes dans votre magasin.
Navigateur	Les données de navigation telles que l'historique ou les archives des recherches.	On a `Browser.SearchColumns` pour les historiques des recherches et `Browser.BookmarkColumns` pour les favoris de l'utilisateur.
Appel	Appels passés, reçus et manqués par l'utilisateur.	On peut trouver ces appels dans `CallLog.Calls`.
Dictionnaire	Les mots que connaît le dictionnaire utilisateur.	Ces mots sont gérés avec `UserDictionary.Words`.

Pour avoir accès à ces contenus natifs, il faut souvent demander une permission. Si vous voulez par exemple accéder aux contacts, n'oubliez pas de demander la permission adaptée : `android.permission.READ_CONTACTS`.

Il existe des API pour vous aider à construire les URI pour les fournisseurs de contenu. Vous connaissez déjà `Uri.Builder`, mais il existe aussi `ContentUris` rien que pour les fournisseurs de contenu. Il contient par exemple la méthode statique `Uri ContentUris.withAppendedId(Uri contentUri, long id)` avec `contentUri` l'URI et `id` l'identifiant de la ligne à récupérer :

```
```

20.1.3. Effectuer des opérations sur un fournisseur de contenu

Vous verrez ici d'énormes ressemblances avec la manipulation des bases de données, c'est normal, les deux API se fondent sur les mêmes principes fondamentaux. Il existe deux objets sur lesquels on peut effectuer les requêtes. Soit directement sur le `ContentResolver`, auquel cas vous devrez fournir à chaque fois l'URI du fournisseur de contenu visé. Soit, si vous effectuez les opérations sur le même fournisseur à chaque fois, vous pouvez utiliser plutôt un `ContentProviderClient`, afin de ne pas avoir à donner l'URI à chaque fois. On peut obtenir un `ContentProviderClient` en faisant `ContentProviderClient acquireContentProviderClient(String name)` sur un `ContentResolver`, `name` étant l'autorité du fournisseur.

Il n'est pas nécessaire de fermer un `ContentResolver`, cependant il faut appliquer `boolean release()` sur un `ContentProviderClient` pour aider le système à libérer de la mémoire. Exemple :

```

```

Les méthodes à utiliser entre les deux objets sont similaires, ils ont les mêmes paramètres, même si `ContentProviderClient` n'a pas besoin qu'on précise d'URI systématiquement. Je ne présenterai d'ailleurs que les méthodes de `ContentProvider`, retenez simplement qu'il suffit de ne pas passer le paramètre de type URI pour utiliser la méthode sur un `ContentProviderClient`.

20.1.3.1. Ajouter des données

Il existe deux méthodes pour ajouter des données. Il y a `Uri insert(Uri url, ContentValues values)`, qui permet d'insérer une valeur avec un `ContentValues` que nous avons appris à utiliser avec les bases de données. L'URI retournée représente la nouvelle ligne insérée.

```

```

Il est aussi possible d'utiliser `int bulkInsert(Uri url, ContentValues[] initialValues)` pour insérer plusieurs valeurs à la fois. Cette méthode retourne le nombre de lignes créées.

20.1.3.2. Récupérer des données

Il n'existe qu'une méthode cette fois : `Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)` avec les mêmes paramètres que d'habitude :

- `uri` indique le fournisseur de contenu.
- `project` est un tableau des colonnes de la table à récupérer.
- `selection` correspond à la valeur du `WHERE`.
- `selectionArgs` permet de remplacer les « ? » dans `selection` par des valeurs.
- `sortOrder` peut valoir « ASC » pour ranger les lignes retournées dans l'ordre croissant et « DESC » pour l'ordre décroissant.

Les résultats sont présentés dans un `Cursor`.

Par exemple, pour récupérer tous les utilisateurs dont le nom est « Apol » on peut faire :

```

```

20.1.3.3. Mettre à jour des données

On utilise `int update(Uri uri, ContentValues values, String where, String[] selectionArgs)` qui retourne le nombre de lignes mises à jour. Par exemple, pour changer le nom du métier « Autre » en « Les autres encore », on fera :

```
```

20.1.3.4. Supprimer des données

Pour cela, il existe `int delete(Uri url, String where, String[] selectionArgs)` qui retourne le nombre de lignes mises à jour. Ainsi, pour supprimer les membres de nom « Apol » et de prénom « Lidore », on fera :

```
```

20.2. Créer un fournisseur

Maintenant que vous savez exploiter les fournisseurs de contenu, on va apprendre à en créer pour que vous puissiez mettre vos bases de données à disposition d'autres applications. Comme je l'ai déjà dit, il n'est pas rare qu'une application n'ait pas de fournisseur, parce qu'on les utilise uniquement pour certaines raisons particulières :

— Vous voulez permettre à d'autres applications d'accéder à des données complexes ou certains fichiers.
— Vous voulez permettre à d'autres applications de pouvoir copier des données complexes qui vous appartiennent.
— Enfin, vous voulez peut-être aussi permettre à d'autres applications de faire des recherches sur vos données complexes.

Une autre raison de ne construire un fournisseur que si nécessaire est qu'il ne s'agit pas d'une tâche triviale : la quantité de travail peut être énorme et la présence d'un fournisseur peut compromettre votre application si vous ne vous protégez pas.

Si vous voulez juste une base de données, n'oubliez pas que vous pouvez très bien le faire sans fournisseur de contenu. Je dis cela parce que certains ont tendance à confondre les deux concepts.

La préparation de la création d'un fournisseur de contenu se fait en plusieurs étapes.

20.2.1. L'URI

Vous l'avez bien compris, pour identifier les données à récupérer, l'utilisateur aura besoin d'une URI. Elle contiendra une autorité afin de permettre la récupération du fournisseur de contenu et un chemin pour permettre d'affiner la sélection et choisir une table, un fichier ou encore une ligne dans une table.

20.2.1.1. Le schéma

Il permet d'identifier quel type de contenu désigne l'URI. Vous le savez déjà, dans le cas des fournisseurs de contenu, ce schéma sera `content://`.

20.2.1.2. L'autorité

Elle sera utilisée comme identifiant pour Android. Quand on déclare un fournisseur dans le Manifest, elle sera inscrite dans un registre qui permettra de la distinguer parmi tous les fournisseurs quand on y fera appel. De manière standard, on utilise le nom du package dans l'autorité afin d'éviter les conflits avec les autres fournisseurs. Ainsi, si le nom de mon package est `sdz.chapitreQuatre.example`, alors pour le fournisseur j'utiliserai `sdz.chapitreQuatre.example.provider`.

20.2.1.3. Le chemin

Il n'y a rien d'obligatoire, mais en général le premier segment de chemin est utilisé pour identifier une table et le second est utilisé comme un identifiant. De ce fait, si on a deux tables `table1` et `table2`, on peut envisager d'y accéder avec `sdz.chapitreQuatre.example.provider/table1` et `sdz.chapitreQuatre.example.provider/table2`. Ensuite, pour avoir le cinquième élément de `table1`, on fait `sdz.chapitreQuatre.example.provider/table1/5` .

Vous pouvez avoir plusieurs segments ou faire en sorte qu'un segment ne corresponde pas à une table, c'est votre choix.

20.2.1.4. UriMatcher

Comme il existe beaucoup d'URI, il va falloir une technique pour toutes les gérer. C'est pourquoi je vais vous apprendre à utiliser `UriMatcher` qui analysera tout seul les URI et prendra les décisions pour vous.

On crée un `UriMatcher` toujours de la même manière :

```
```

Cependant on n'utilisera qu'un seul `UriMatcher` par classe, alors on le déclarera en tant qu'attribut de type `static final` :

On va ensuite ajouter les différentes URI que pourra accepter le fournisseur, et on associera à chacune de ces URI un identifiant. De cette manière, on donnera des URI à notre `UriMatcher` et il déterminera tout seul le type de données associé.

Pour ajouter une URI, on utilise `void addURI(String authority, String path, int code)`, avec l'autorité dans `authority`, `path` qui incarne le chemin (on peut mettre `#` pour symboliser un nombre et `*` pour remplacer une quelconque chaîne de caractères) et enfin `code` l'identifiant associé à l'URI. De plus, comme notre `UriMatcher` est statique, on utilise ces ajouts dans un bloc `static` dans la déclaration de notre classe :

Enfin, on vérifie si une URI correspond aux filtres installés avec `int match(Uri uri)`, la valeur retournée étant l'identifiant de l'URI analysée :

20.2.2. Le type MIME

Android a besoin de connaître le type MIME des données auxquelles donne accès votre fournisseur de contenu, afin d'y accéder sans avoir à préciser leur structure ou leur implémentation. On a de ce fait besoin d'une méthode qui indique ce type (`String getType(Uri uri)`) dont le retour est une chaîne de caractères qui contient ce type MIME.

Cette méthode devra être capable de retourner deux formes de la même valeur en fonction de ce que veut l'utilisateur : une seule valeur ou une collection de valeurs. En effet, vous vous souvenez, un type MIME qui n'est pas officiel doit prendre sous Android la forme `vnd.android.cursor.X` avec `X` qui vaut `item` pour une ligne unique et `dir` pour une collection de lignes. Il faut ensuite une chaîne qui définira le type en lui-même, qui doit respecter la forme `vnd.<nom unique>.<type>`.

Voici ce que j'ai choisi :

C'est ici que l'`UriMatcher` prendra tout son intérêt :

20.2.3. Le stockage

Comment allez-vous stocker les données ? En général, on utilise une base de données, mais vous pouvez très bien opter pour un stockage sur support externe. Je vais me concentrer ici sur l'utilisation des bases de données.

On va avoir une classe qui représente la base de données et, à l'intérieur de cette classe, des *classes internes constantes* qui représenteront chaque table. Une classe constante est une classe déclarée avec les modificateurs `static final`. Cette classe contiendra des attributs constants (donc qui possèdent aussi les attributs `static final)` qui définissent les URI, le nom de la table, le nom de ses colonnes, les types MIME ainsi que toutes les autres données nécessaires à l'utilisation du fournisseur. L'objectif de cette classe, c'est d'être certains que les applications qui feront appel au fournisseur pourront le manipuler aisément, même si certains changements sont effectués au niveau de la valeur des URI, du nom des colonnes ou quoi que ce soit d'autre. De plus, les classes constantes aident les développeurs puisque les constantes ont des noms mnémoniques plus pratiques à utiliser que si on devait retenir toutes les valeurs.

Bien entendu, comme les développeurs n'auront pas accès au code en lui-même, c'est à vous de bien documenter le code pour qu'ils puissent utiliser vos fournisseurs de contenu.

```

```

Comme vous pouvez le voir, ma classe `Metier` dérive de `BaseColumns`. Il s'agit d'une petite classe qui définit deux attributs indispensables : `_ID` (qui représente l'identifiant d'une ligne) et `_COUNT` (qui représente le nombre de lignes dans une requête).

20.2.4. Le Manifest

Chaque fournisseur de contenu s'enregistre sur un appareil à l'aide du Manifest. On aura besoin de préciser une autorité ainsi qu'un identifiant et la combinaison des deux se doit d'être unique. Cette combinaison n'est que la base utilisée pour constituer les requêtes de contenu. Le nœud doit être de type `provider`, puis on verra ensuite deux attributs : `android:name` pour le nom du composant (comme pour tous les composants) et `android:authorities` pour l'autorité.

```

```

20.2.5. La programmation

On fait dériver une classe de `ContentProvider` pour gérer les requêtes qui vont s'effectuer sur notre fournisseur de contenu. Chaque opération qu'effectuera une application sur votre fournisseur de contenu sera à gérer dans la méthode idoine. Je vais donc vous présenter le détail de chaque méthode.

20.2.5.1. boolean onCreate()

Cette méthode de *callback* est appelée automatiquement dès qu'un `ContentResolver` essaie d'y accéder pour la première fois.

Le plus important ici est d'éviter les opérations qui prennent du temps, puisqu'il s'agit du démarrage, sinon celui-ci durera trop longtemps. Je pense par exemple à éviter les initialisations qui pourraient prendre du temps (comme créer, ouvrir, mettre à jour ou analyser la base de données), de façon à permettre aux applications de se lancer plus vite, d'éviter les efforts inutiles si le fournisseur n'est pas nécessaire, d'empêcher les erreurs de base de données (comme par exemple un disque plein), ou d'arrêter le lancement de l'application. De ce fait, faites en sorte de ne jamais appeler `getReadableDatabase()` ou `getWritableDatabase()` dans cette méthode.

La meilleure chose à faire, est d'implémenter `onOpen(SQLiteDatabase)` comme nous avons appris à le faire, pour initialiser la base de données quand elle est ouverte pour la première fois (dès que le fournisseur reçoit une quelconque requête concernant la base).

Par exemple, vous pouvez créer un `SQLiteOpenHelper` dans `onCreate()`, mais ne créez les tables que la première fois que vous ouvrez vraiment la base. Rappelez-vous que la première fois que vous appelez `getWritableDatabase()` on fera automatiquement appel à `onCreate()` de `SQLiteOpenHelper`.

N'oubliez pas de retourner `true` si tout s'est bien déroulé.

```

```

20.2.5.2. Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)

Permet d'effectuer des recherches sur la base. Elle doit retourner un `Cursor` qui contient le résultat de la recherche ou doit lancer une exception en cas de problème. S'il n'y a pas de résultat qui correspond à la recherche, alors il faut renvoyer un `Cursor` vide, et non `null`, qui est plutôt réservé aux erreurs.

```

```

20.2.5.3. Uri insert(Uri uri, ContentValues values)

On l'utilise pour insérer des données dans le fournisseur. Elle doit retourner l'URI de la nouvelle ligne. Comme vous le savez déjà, ce type d'URI doit être constitué de l'URI qui caractérise la table suivie de l'identifiant de la ligne.

Afin d'alerter les éventuels observateurs qui suivent le fournisseur, on indique que l'ensemble des données a changé avec la méthode `void notifyChange(Uri uri, ContentObserver observer)`, `uri` indiquant les données qui ont changé et `observer` valant `null`.

20.2.5.4. int update(Uri uri, ContentValues values, String selection, String[] selectionArgs)

Met à jour des données dans le fournisseur. Il faut retourner le nombre de lignes modifiées. N'oubliez pas d'alerter les observateurs avec `notifyChange()` encore une fois.

20.2.5.5. int delete(Uri uri, String selection, String[] selectionArgs)

Supprime des éléments du fournisseur et doit retourner le nombre de lignes supprimées. Pour alerter les observateurs, utilisez encore une fois `void notifyChange(Uri uri, ContentOb server observer)`.

20.2.5.6. String getType(Uri uri)

Retourne le type MIME des données concernées par `uri`. Vous connaissez déjà cette méthode par cœur !

- Les fournisseurs de contenu permettent de rendre accessibles les données d'une application sans connaitre son moyen de stockage.
- Pour accéder à un fournisseur de contenu, vous êtes obligés de passer par un objet client `ContentResolver`.
- L'URI pour un fournisseur de contenu est sous la forme suivante : un schéma et l'information :
  - Le schéma d'un fournisseur de contenu est `content` et s'écrit `content://`.
  - L'information est un chemin qui devient de plus en plus précis à force de rentrer dans des parties spécifiques. La partie de l'information qui permet de pointer de manière unique vers le bon fournisseur de contenu est l'autorité. Quant à l'affinement de la requête par des "/", cela s'appelle des segments.
- Il n'est pas rare qu'une application n'offre pas de fournisseur de contenus. Il y a plusieurs raisons pour lesquelles vous pourrez en développer un :
  - Vous voulez permettre à d'autres applications d'accéder à des données complexes ou certains fichiers.

— Vous voulez permettre à d'autres applications de pouvoir copier des données complexes qui vous appartiennent.
— Vous voulez peut-être aussi permettre à d'autres applications de faire des recherches sur vos données complexes.

21. Créer un AppWidget

Comme vous le savez probablement, une des forces d'Android est son côté personnalisable. Un des exemples les plus probants est qu'il est tout à fait possible de choisir les éléments qui se trouvent sur l'écran d'accueil. On y trouve principalement des icônes, mais les utilisateurs d'Android sont aussi friands de ce qu'on appelle les « AppWidgets », applications miniatures destinées à être utilisées dans d'autres applications. Ce AppWidgets permettent d'améliorer une application à peu de frais en lui ajoutant un compagnon permanent. De plus, mettre un AppWidget sur son écran d'accueil permet à l'utilisateur de se rappeler l'existence de votre produit et par conséquent d'y accéder plus régulièrement. Par ailleurs, les AppWidgets peuvent accorder un accès direct à certaines fonctionnalités de l'application sans avoir à l'ouvrir, ou même ouvrir l'application ou des portions de l'application.

Un AppWidget est divisé en plusieurs unités, toutes nécessaires pour fonctionner. On retrouve tout d'abord une interface graphique qui détermine quelles sont les vues qui le composent et leurs dispositions. Ensuite, un élément gère le cycle de vie de l'AppWidget et fait le lien entre l'AppWidget et le système. Enfin, un dernier élément est utilisé pour indiquer les différentes informations de configuration qui déterminent certains aspects du comportement de l'AppWidget. Nous allons voir tous ces éléments, comment les créer et les manipuler.

21.1. L'interface graphique

La première chose à faire est de penser à l'interface graphique qui représentera la mise en page de l'AppWidget. Avant de vous y mettre, n'oubliez pas de réfléchir un peu. Si votre AppWidget est l'extension d'une application, faites en sorte de respecter la même charte graphique de manière à assurer une véritable continuité dans l'utilisation des deux programmes. Le pire serait qu'un utilisateur ne reconnaisse pas votre application en voyant un AppWidget et n'arrive pas à associer les deux dans sa tête.

Vous allez comme d'habitude devoir créer un layout dans le répertoire `res/layout/`. Cependant, il ne peut pas contenir toutes les vues qui existent. Voici les layouts acceptés :

- `FrameLayout`
- `LinearLayout`
- `RelativeLayout`

… ainsi que les widgets acceptés :

- `AnalogClock`
- `Button`
- `Chronometer`
- `ImageButton`
- `ImageView`

— `ProgressBar`
— `TextView`

Ne confondez pas les widgets, ces vues qui ne peuvent pas contenir d'autres vues, et les AppWidgets. Pour être franc, vous trouverez le terme « widget » utilisé pour désigner des AppWidgets, ce qui est tout à fait correct, mais pour des raisons pédagogiques je vais utiliser AppWidget.

Pourquoi uniquement ces vues-là ?

Toutes les vues ne sont pas égales. Ces vues-là sont des `RemoteViews`, c'est-à-dire qu'on peut y avoir accès quand elles se trouvent dans un autre processus que celui dans lequel on travaille. Au lieu de désérialiser une hiérarchie de vues comme on le fait d'habitude, on désérialisera un layout dans un objet de type `RemoteViews`. Il est ainsi possible de configurer les vues dans notre receiver pour les rendre accessibles à une autre activité, celle de l'écran d'accueil par exemple.

L'une des contreparties de cette technique est que vous ne pouvez pas implémenter facilement la gestion des évènements avec un `OnClickListener` par exemple. À la place, on va attribuer un `PendingIntent` à notre `RemoteViews` de façon à ce qu'il sache ce qu'il doit faire en cas de clic, mais nous le verrons plus en détail bientôt.

Enfin, sachez qu'on ne retient pas de référence à des `RemoteViews`, tout comme on essaie de ne jamais faire de référence à des `context`.

Voici un exemple standard :

```
```

Vous remarquerez que j'ai utilisé des valeurs bien précises pour le bouton. En effet, il faut savoir que l'écran d'accueil est divisé en cellules. Une cellule est l'unité de base de longueur dans cette application, par exemple une icône fait une cellule de hauteur et une cellule de largeur. La plupart des écrans possèdent quatre cellules en hauteur et quatre cellules en largeur, ce qui donne $4 \times 4 = 16$ cellules en tout.

Pour déterminer la mesure que vous désirez en cellules, il suffit de faire le calcul $(74 \times N) - 2$ avec `N` le nombre de cellules voulues. Ainsi, j'ai voulu que mon bouton fasse une cellule de hauteur, ce qui donne $(74 \times 1) - 2 = 72$ dp.

Vous vous rappelez encore les `dp` ? C'est une unité qui est proportionnelle à la résolution de l'écran, contrairement à d'autres unités comme le pixel par exemple. Imaginez, sur un écran qui a 300 pixels en longueur, une ligne qui fait 150 pixels prendra la moitié de l'écran, mais sur un écran qui fait 1500 pixels de longueur elle n'en fera qu'un dixième ! En revanche, avec les dp (ou dip, c'est pareil), Android calculera automatiquement la valeur en pixels pour adapter la taille de la ligne à la résolution de l'écran.

21.2. Définir les propriétés

Maintenant, il faut préciser différents paramètres de l'AppWidget dans un fichier XML. Ce fichier XML représente un objet de type `AppWidgetProviderInfo`.

Tout d'abord, la racine est de type `<appwidget-provider>` et doit définir l'espace de nommage `android`, comme ceci :

```
```

Vous pouvez définir la hauteur minimale de l'AppWidget avec `android:minHeight` et sa largeur minimale avec `android:minWidth`. Les valeurs à indiquer sont en dp comme pour le layout.

Ensuite, on utilise `android:updatePeriodMillis` pour définir la fréquence de mise à jour voulue, en millisecondes. Ainsi, `android:updatePeriodMillis="60000"` fait une minute, `android:updatePeriodMillis="3600000"` fait une heure, etc. Puis on utilise `android:initialLayout` pour indiquer la référence au fichier XML qui indique le layout de l'AppWidget. Enfin, vous pouvez associer une activité qui permettra de configurer l'AppWidget avec `android:configure`.

Voici par exemple ce qu'on peut trouver dans un fichier du genre `res/xml/appwidget_info.xml` :

```
```

21.3. Le code

21.3.1. Le receiver

Le composant de base qui permettra l'interaction avec l'AppWidget est `AppWidgetProvider`. Il permet de gérer tous les évènements autour de la vie de l'AppWidget. `AppWidgetProvider` est une classe qui dérive de `BroadcastReceiver`, elle va donc recevoir les divers broadcast intents qui sont émis *et* qui sont destinés à l'AppWidget. On retrouve quatre évènements pris en compte : l'activation, la mise à jour, la désactivation et la suppression. Comme d'habitude, chaque période de la vie d'un AppWidget est représentée par une méthode de *callback*.

21.3.1.1. La mise à jour

La méthode la plus importante est celle relative à la mise à jour, vous devrez l'implémenter chaque fois. Il s'agit de `public void onUpdate(Context context, AppWidgetManager appWidgetManager, int[] appWidgetIds)` avec comme paramètres :

— Le `context` dans lequel le receiver s'exécute.

— `appWidgetManager` représente le gestionnaire des AppWidgets, il permet d'avoir des informations sur tous les AppWidgets disponibles sur le périphérique et de les mettre à jour.
— Les identifiants des AppWidgets à mettre à jour sont contenus dans `appWidgetIds`.

Cette méthode sera appelée à chaque expiration du délai `updatePeriodMillis`.

Ainsi, dans cette méthode, on va récupérer l'arbre de `RemoteViews` qui constitue l'interface graphique et on mettra à jour les vues qui ont besoin d'être mises à jour. Pour récupérer un `RemoteViews`, on utilisera le constructeur `RemoteViews(String packageName, int layoutId)` qui a besoin du nom du package du `context` dans `packageName` (on le récupère facilement avec la méthode `String getPackageName()` de `Context`) et l'identifiant du layout dans `layoutId`.

Vous pouvez ensuite manipuler n'importe quelle vue qui se trouve dans cette hiérarchie à l'aide de diverses méthodes de manipulation. Par exemple, pour changer le texte d'un `TextView`, on fera `void setTextViewText(int viewId, CharSequence text)` avec `viewId` l'identifiant du `TextView` et le nouveau `text`. Il n'existe bien entendu pas de méthodes pour toutes les méthodes que peuvent exécuter les différentes vues, c'est pourquoi `RemoteViews` propose des méthodes plus génériques qui permettent d'appeler des méthodes sur les vues et de leur passer des paramètres. Par exemple, un équivalent à :

```

```

… pourrait être :

```

```

On a en fait fait appel à la méthode `setText` de `TextView` en lui passant un `String`.

Maintenant que les modifications ont été faites, il faut les appliquer. En effet, elles ne sont pas effectives toutes seules, il vous faudra utiliser la méthode `void updateAppWidget(int appWidgetId, RemoteViews views)` avec `appWidgetId` l'identifiant du widget qui contient les vues et `views` la racine de type `RemoteViews`.

```

```

21.3.1.2. Les autres méthodes

Tout d'abord, comme `AppWidgetProvider` dérive de `BroadcastReceiver`, on pourra retrouver les méthodes de `BroadcastReceiver`, dont `public void onReceive(Context context, Intent intent)` qui est activé dès qu'on reçoit un broadcast intent.

La méthode `public void onEnabled(Context context)` n'est appelée que la première fois qu'un AppWidget est créé. Si l'utilisateur place deux fois un AppWidget sur l'écran d'accueil, alors cette méthode ne sera appelée que la première fois. Le broadcast intent associé est `APP_WIDGET_ENABLED`.

Ensuite, la méthode `public void onDeleted(Context context, int[] appWidgetIds)` est appelée à chaque fois qu'un AppWidget est supprimé. Il répond au broadcast intent `APP_WIDGET_DELETED`.

Et pour finir, quand la toute dernière instance de votre AppWidget est supprimée, le broadcast intent APP_WIDGET_DISABLED est envoyé afin de déclencher la méthode `public void onDisabled(Context context)`.

21.3.2. L'activité de configuration

C'est très simple, il suffit de créer une classe qui dérive de `PreferenceActivity` comme vous savez déjà le faire.

21.4. Déclarer l'AppWidget dans le Manifest

Le composant de base qui représente votre application est le `AppWidgetProvider`, c'est donc lui qu'il faut déclarer dans le Manifest. Comme `AppWidgetProvider` dérive de `BroadcastReceiver`, il faut déclarer un nœud de type `<receiver>`. Cependant, contrairement à un `BroadcastReceiver` classique où l'on pouvait ignorer les attributs `android:icon` et `android:label`, ici il vaut mieux les déclarer. En effet, ils sont utilisés pour donner des informations sur l'écran de sélection des widgets :

```

```

Il faut bien entendu rajouter des filtres à intents dans ce receiver, sinon il ne se lancera jamais. Le seul broadcast intent qui nous intéressera toujours est `android.appwidget.action.APPWIDGET_UPDATE` qui est envoyé à chaque fois qu'il faut mettre à jour l'AppWidget :

```

```

Ensuite, pour définir l'`AppWidgetProviderInfo`, il faut utiliser un élément de type `<meta-data>` avec les attributs `android:name` qui vaut `android.appwidget.provider` et `android:resource` qui est une référence au fichier XML qui contient l'`AppWidgetProviderInfo` :

```

```

Ce qui donne au complet :

```

```

21.5. Application : un AppWidget pour accéder aux tutoriels du Site du Zéro

On va créer un AppWidget qui ne sera pas lié à une application. Il permettra de choisir quel tutoriel du Site du Zéro l'utilisateur souhaite visualiser.

21.5.1. Résultat attendu

Mon AppWidget ressemble à la figure suivante. Évidemment, vous pouvez modifier le design pour obtenir quelque chose de plus… esthétique. Là, c'est juste pour l'exemple.

FIGURE 21.1. – Cet AppWidget permet d'accéder à des tutoriels du Site du Zéro

On peut cliquer sur le titre du tutoriel pour lancer le tutoriel dans un navigateur. Les deux boutons permettent de naviguer dans la liste des tutoriels disponibles.

21.5.2. Aspect technique

Pour permettre aux trois boutons (celui qui affiche le titre est aussi un bouton) de réagir aux clics, on va utiliser la méthode `void setOnClickPendingIntent(int viewId, PendingIntent pendingIntent)` de `RemoteViews` avec `viewId` l'identifiant du bouton et `pendingIntent` le `PendingIntent` qui contient l'`Intent` qui sera exécuté en cas de clic.

Détail important : pour ajouter plusieurs évènements de ce type, il faut différencier chaque `Intent` en leur ajoutant un champ `Données` différent. Par exemple, j'ai rajouté des données de cette manière à mes intents : `intent.setData(Uri.withAppendedPath(Uri.parse("WIDGET://widget/id/"), String.valueOf(Identifiant_de_cette_vue)))`. Ainsi, j'obtiens des données différentes pour chaque intent, même si ces données ne veulent rien dire.

Afin de faire en sorte qu'un intent lance la mise à jour de l'AppWidget, on lui mettra comme action `AppWidgetManager.ACTION_APPWIDGET_UPDATE` et comme extra les identifiants des widgets à mettre à jour ; l'identifiant de cet extra sera `AppWidgetManager.EXTRA_APPWIDGET_ID` :

```

```

Comme `AppWidgetProvider` dérive de `BroadcastReceiver`, vous pouvez implémenter `void onReceive(Context context, Intent intent)` pour gérer chaque intent qui lance ce receiver.

21.5.3. Ma solution

Tout d'abord je déclare mon layout :

```

```

La seule chose réellement remarquable est que le fond du premier bouton est transparent grâce à l'attribut `android:background="@android:color/transparent"`.

Une fois mon interface graphique créée, je déclare mon `AppWidgetProviderInfo` :

```

```

Je désire qu'il fasse au moins 2 cases en hauteur et 2 cases en largeur, et qu'il se rafraîchisse toutes les heures.

J'ai ensuite créé une classe très simple pour représenter les tutoriels :

```

```

Puis, le receiver associé à mon AppWidget :

```

```

Enfin, on déclare le tout dans le Manifest :

```

```

— Un AppWidget est une extension de votre application. Afin que l'utilisateur ne soit pas désorienté, adoptez la même charte graphique que votre application.
— Les seules vues utilisables pour un widget sont les vues `RemoteViews`.
— La déclaration d'un AppWidget se fait dans un élément `appwidget-provider` à partir d'un fichier XML `AppWidgetProviderInfo`.
— La super classe de notre AppWidget sera un `AppWidgetProvider`. Il s'occupera de gérer tous les évènements sur le cycle de vie de notre AppWidget. Cette classe dérive de `BroadcastReceiver`, elle va donc recevoir les divers broadcast intents qui sont émis et qui sont destinés à l'AppWidget.
— Pour déclarer notre AppWidget dans le manifest, nous allons créer un élément `receiver` auquel nous ajoutons un élément `intent-filter`pour lancer notre AppWidget et un élément `meta-data` pour définir l'`AppWidgetProviderInfo`.

Cinquième partie

Exploiter les fonctionnalités d'Android

22. La connectivité réseau

Maintenant que vous savez tout ce qu'il y a à savoir sur les différentes facettes des applications Android, voyons maintenant ce que nous offre notre terminal en matière de fonctionnalités. La première sur laquelle nous allons nous pencher est la connectivité réseau, en particulier l'accès à internet. On va ainsi voir comment surveiller la connexion au réseau ainsi que comment contrôler cet accès. Afin de se connecter à internet, le terminal peut utiliser deux interfaces. Soit le réseau mobile (3G, 4G, etc.), soit le WiFi.

Il y a de fortes chances pour que ce chapitre vous soit utile, puisque statistiquement la permission la plus demandée est celle qui permet de se connecter à internet.

22.1. Surveiller le réseau

Avant toute chose, nous devons nous assurer que l'appareil a bien accès à internet. Pour cela, nous avons besoin de demander la permission au système dans le Manifest :

```
```

Il existe deux classes qui permettent d'obtenir des informations sur l'état du réseau. Si vous voulez des informations sur sa disponibilité de manière générale, utilisez `ConnectivityManager`. En revanche, si vous souhaitez des informations sur l'état de l'une des interfaces réseau (en général le réseau mobile ou le WiFi), alors utilisez plutôt `NetworkInfo`.

On peut récupérer le gestionnaire de connexions dans un `Context` avec `ConnectivityManager Context.getSystemService(Context.CONNECTIVITY_SERVICE)`.

Ensuite, pour savoir quelle est l'interface active, on peut utiliser la méthode `NetworkInfo getActiveNetworkInfo()`. Si aucun réseau n'est disponible, cette méthode renverra `null`.

Vous pouvez aussi vérifier l'état de chaque interface avec `NetworkInfo getNetworkInfo(ConnectivityManager.TYPE_WIFI)` ou `NetworkInfo getNetworkInfo(ConnectivityManager.TYPE_MOBILE)` pour le réseau mobile.

Enfin, il est possible de demander à un `NetworkInfo` s'il est connecté à l'aide de la méthode `boolean isAvailable()` :

```
```

De manière générale, on préférera utiliser internet si l'utilisateur est en WiFi parce que le réseau mobile est plus lent et est souvent payant. Il est conseillé de mettre en garde l'utilisateur avant de télécharger quelque chose en réseau mobile. Vous pouvez aussi envisager de bloquer les téléchargements quand seul le réseau mobile est disponible, comme c'est souvent fait.

Il est cependant possible que l'état de la connexion change et qu'il vous faille réagir à ce changement. Dès qu'un changement surgit, le broadcast intent `ConnectivityManager.CONNECTIVITY_ACTION` est envoyé (sa valeur est étrangement `android.net.conn.CONNECTIVITY_CHANGE`). Vous pourrez donc l'écouter avec un receiver déclaré de cette manière :

```

```

Vous trouverez ensuite dans les extras de l'intent plus d'informations. Par exemple `ConnectivityManager.EXTRA_NO_CONNECTIVITY` renvoie un booléen qui vaut `true` s'il n'y a pas de connexion à internet en cours. Vous pouvez aussi obtenir directement un `NetworkInfo` avec l'extra `ConnectivityManager.EXTRA_OTHER_NETWORK_INFO` afin d'avoir plus d'informations sur le changement.

22.2. Afficher des pages Web

Il pourrait vous prendre l'envie de montrer à votre utilisateur une page Web. Ou alors il se peut que vous vouliez faire une interface graphique à l'aide de HTML. Nous avons déjà vu une méthode pour mettre du HTML dans des `TextView`, mais ces méthodes ne sont pas valides pour des utilisations plus poussées du HTML, comme par exemple pour afficher des images ; alors pour afficher une page complète, n'imaginez même pas.

Ainsi, pour avoir une utilisation plus poussée de HTML, on va utiliser une nouvelle vue qui s'appelle `WebView`. En plus d'être une vue très puissante, `WebView` est commandé par WebKit, un moteur de rendu de page Web qui fournit des méthodes pratiques pour récupérer des pages sur internet, effectuer des recherches dans la page, etc.

22.2.1. Charger directement du HTML

Pour insérer des données HTML sous forme textuelle, vous pouvez utiliser `void loadData(String data, String mimeType, String encoding)` avec `data` les données HTML, `mimeType` le type MIME (en général `text/html`) et l'encodage défini dans `encoding`. Si vous ne savez pas quoi mettre pour `encoding`, mettez « UTF-8 », cela devrait aller la plupart du temps.

```

```

On obtient alors la figure suivante.

FIGURE 22.1. – Du HTML s'affiche

N'oubliez pas de préciser l'encodage dans l'en-tête, sinon vos accents ne passeront pas.

22.2.2. Charger une page sur internet

La première chose à faire est de demander la permission pour aller sur internet dans le Manifest :

Puis vous pouvez charger le contenu avec `void loadUrl(String url)`. Ainsi, avec ce code :

… on obtient la figure suivante :

FIGURE 22.2. – Le Site du Zéro est affiché à l'écran

22.3. Effectuer des requêtes HTTP

22.3.1. Rappels sur le protocole HTTP

HTTP est un protocole de communication, c'est-à-dire un ensemble de règles à suivre quand deux machines veulent communiquer. On l'utilise surtout dans le cadre du *World Wide Web*, une des applications d'internet, celle qui vous permet de voir des sites en ligne. Vous remarquerez d'ailleurs que l'URI que vous utilisez pour accéder à un site sur internet a pour schéma `http:`, comme sur cette adresse : `http://www.siteduzero.com`.

Il fonctionne de cette manière : un client envoie une requête HTTP à un serveur qui va réagir et répondre en HTTP en fonction de cette entrée, comme le montre la figure suivante.

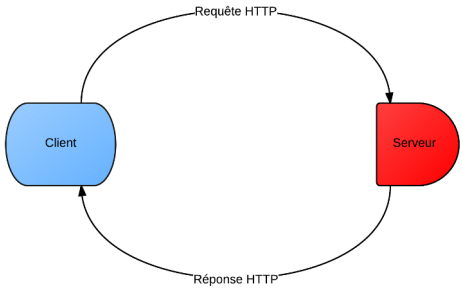

FIGURE 22.3. – La requête et la réponse utilisent le même protocole, mais leur contenu est déterminé par le client ou le serveur

Il existe plusieurs méthodes en HTTP. Ce sont des commandes, des ordres qui accompagnent les requêtes. Par exemple, si on veut récupérer une ressource on utilise la méthode `GET`. Quand vous tapez une adresse dans la barre de navigation de votre navigateur internet, il fera un `GET` pour récupérer le contenu de la page.

À l'opposé de `GET`, on trouve `POST` qui est utilisé pour envoyer des informations. Quand vous vous inscrivez sur un site, ce qui se fait souvent à l'aide d'un formulaire, l'envoi de ce dernier correspond en fait à un `POST` vers le serveur qui contient les diverses informations que vous avez envoyées. Mais comment ça fonctionne, concrètement ? C'est simple, dans votre requête `POST`, votre navigateur va ajouter comme données un ensemble de couples identifiant-clé de cette forme-ci : `identifiant1=clé1&identifiant2=clé2&identifiant3=clé3`. Ainsi, votre serveur sera capable de retrouver les identifiants avec les clés qui y sont associées.

22.3.2. Le HTTP sous Android

Il existe deux méthodes pour manipuler le protocole HTTP sous Android. La première est fournie par Apache et est celle que vous êtes censés utiliser avant l'API 9 (Gingerbread). En pratique, nous allons voir l'autre méthode même si elle n'est pas recommandée pour l'API 7, parce qu'elle est à privilégier pour la plupart de vos applications.

Pour commencer, la première chose à faire est de savoir sur quelle URL on va opérer avec un objet de type `URL`. La manière la plus simple d'en créer un est de le faire à l'aide d'une chaîne de caractères :

Toutes vos requêtes HTTP devront se faire dans un thread différent du thread UI, puisqu'il s'agit de processus lents qui risqueraient d'affecter les performances de votre application.

On peut ensuite ouvrir une connexion vers cette URL avec la méthode `URLConnection openConnection()`. Elle renvoie un `URLConnection`, qui est une classe permettant de lire et d'écrire depuis une URL. Ici, nous allons voir en particulier la connexion à une URL avec le protocole HTTP, on va donc utiliser une classe qui dérive de `URLConnection` : `HttpURLConnection`.

Il est ensuite possible de récupérer le flux afin de lire des données en utilisant la méthode `InputStream getInputStream()`. Avant cela, vous souhaiterez peut être vérifier le code de réponse fourni par le serveur HTTP, car votre application ne réagira pas de la même manière si vous recevez une erreur ou si tout s'est déroulé correctement. Vous pouvez le faire avec la méthode `int getResponseCode()` :

Enfin, si vous voulez effectuer des requêtes sortantes, c'est-à-dire vers un serveur, il faudra utiliser la méthode `setDoOutput(true)` sur votre `HttpURLConnection` afin d'autoriser les flux sortants. Ensuite, si vous connaissez la taille des paquets que vous allez transmettre, utilisez `void setFixedLengthStreamingMode(int contentLength)` pour optimiser la procédure, avec `contentLength` la taille des paquets. En revanche, si vous ne connaissez pas cette taille, alors utilisez `setChunkedStreamingMode(0)` qui va séparer votre requête en paquets d'une taille définie par le système :

Dans les versions les plus récentes d'Android, effectuer des requêtes HTTP dans le thread UI soulèvera une exception, vous serez donc obligés de créer un thread pour effectuer vos requêtes. De toute manière, même si vous êtes dans une ancienne version qui ne soulève pas d'exception, il vous faut quand même créer un nouveau thread, parce que c'est la bonne manière.

Pour finir, comme pour n'importe quel autre flux, n'oubliez pas de vous déconnecter avec `void disconnect()`.

Avant de vous laissez, je vais vous montrer une utilisation correcte de cette classe. Vous vous rappelez que je vous avais dit que normalement il ne fallait pas utiliser cette API pour faire ces requêtes ; c'est en fait parce qu'elle est boguée. L'un des bugs qui vous agacera le plus est que,

vous aurez beau demander de fermer un flux, Android ne le fera pas. Pour passer outre, nous allons désactiver une fonctionnalité du système qui permet de contourner le problème :

```
```

Voici par exemple une petite application qui envoie des données à une adresse et récupère ensuite la réponse :

```
```

— Dans votre vie de programmeur Android, il est très probable que vous liez au moins une application à internet tellement c'est quelque chose de courant.
— On peut obtenir des informations sur l'état de la connectivité de l'appareil grâce à `ConnectivityManager`. C'est indispensable parce qu'il y a des chances que l'utilisateur passe du Wifi à un réseau mobile lorsqu'il exploite votre application. Dès que ça arrive, il faut couper tout téléchargement pour que votre pauvre utilisateur ne se retrouve pas avec une facture longue comme son bras !
— On peut très facilement afficher du code HTML avec une `WebView`.
— Sur le web, pour envoyer et recevoir des applications, on utilise le protocole HTTP. Il possède en autre la méthode `GET` pour récupérer du contenu sur internet et la méthode `POST` pour en envoyer.
— Quand on récupère du contenu sur internet, on passe toujours par un thread, car récupérer cela prend du temps et impacterait trop l'utilisateur si on l'employait dans le thread UI.
— On récupère et on poste du contenu facilement à l'aide de flux comme nous l'avons toujours fait pour écrire dans un fichier.

23. Apprenez à dessiner

Je vous propose d'approfondir nos connaissances du dessin sous Android. Même si dessiner quand on programme peut sembler trivial à beaucoup d'entre vous, il faut que vous compreniez que c'est un élément qu'on retrouve dans énormément de domaines de l'informatique. Par exemple, quand on veut faire sa propre vue, on a besoin de la dessiner. De même, dessiner est une étape essentielle pour faire un jeu.

Enfin, ne vous emballez pas parce que je parle de jeu. En effet, un jeu est bien plus que des graphismes, il faut créer différents moteurs pour gérer le *gameplay*, il faut travailler sur l'aspect sonore, etc. De plus, la méthode présentée ici est assez peu adaptée au jeu. Mais elle va quand même nous permettre de faire des choses plutôt sympa.

23.1. La toile

Non, non, je ne parle pas d'internet ou d'un écran de cinéma, mais bien d'une vraie toile. Pas en lin ni en coton, mais une toile de pixels. C'est sur cette toile que s'effectuent nos dessins. Et vous l'avez déjà rencontrée, cette toile ! Mais oui, quand nous dessinions nos propres vues, nous avons vu un objet de type `Canvas` sur lequel dessiner !

Pour être tout à fait franc, ce n'était pas exactement la réalité. En effet, on ne dessine pas sur un `Canvas`, ce n'est pas un objet graphique, mais une interface qui va dessiner sur un objet graphique. Le dessin est en fait effectué sur un `Bitmap`. Ainsi, il ne suffit pas de créer un `Canvas` pour pouvoir dessiner, il faut lui attribuer un `Bitmap`.

La plupart du temps, vous n'aurez pas besoin de donner de `Bitmap` à un `Canvas` puisque les `Canvas` qu'on vous donnera auront déjà un `Bitmap` associé. Les seuls moments où vous devrez le faire manuellement sont les moments où vous créerez vous-mêmes un `Canvas`.

Ainsi, un `Canvas` est un objet qui réalise un dessin et un `Bitmap` est une surface sur laquelle dessiner. Pour raisonner par analogie, on peut se dire qu'un `Canvas` est un peintre et un `Bitmap` une toile. Cependant, que serait un peintre sans son fidèle pinceau ? Un pinceau est représenté par un objet `Paint` et permet de définir la couleur du trait, sa taille, etc. Alors quel est votre rôle à vous ? Eh bien, imaginez-vous en tant que client qui demande au peintre (`Canvas`) de dessiner ce que vous voulez, avec la couleur que vous voulez et sur la surface que vous voulez. C'est donc au `Canvas` que vous donnerez des ordres pour dessiner.

23.1.1. La toile

Il n'y a pas grand-chose à savoir sur les `Bitmap`. Tout d'abord, il n'y a pas de constructeur dans la classe `Bitmap`. Le moyen le plus simple de créer un `Bitmap` est de passer par la méthode statique `Bitmap createBitmap(int width, int height, Bitmap.Config config)` avec `width` la largeur de la surface, `height` sa hauteur et `config` un objet permettant de déterminer comment les pixels seront stockés dans le `Bitmap`.

En fait, le paramètre `config` permet de décrire quel espace de couleur sera utilisé. En effet, les couleurs peuvent être représentées d'au moins trois manières :

— Pour que chaque pixel ne puisse être qu'une couleur, utilisez `Bitmap.Config.RGB_565`.
— Pour que chaque pixel puisse être soit une couleur, soit transparent (c'est-à-dire qu'il n'affiche pas de couleur), utilisez `Bitmap.Config.ARGB_8888`.
— Enfin, si vous voulez que seul le canal qui représente des pixels transparents soit disponible, donc pour n'avoir que des pixels transparents, utilisez `Bitmap.Config.ALPHA_8`.

Par exemple :

```
```

Il existe aussi une classe dédiée à la construction de `Bitmap` : `BitmapFactory`. Ainsi, pour créer un `Bitmap` depuis un fichier d'image, on fait `BitmapFactory.decodeFile("Chemin vers le fichier")`. Pour le faire depuis un fichier de ressource, on utilise la méthode statique `decodeResource(Resources ressources, int id)` avec le fichier qui permet l'accès aux `ressources` et l'identifiant de la ressource dans `id`. Par exemple :

```
```

N'oubliez pas qu'on peut récupérer un fichier de type `Resources` sur n'importe quel `Context` avec la méthode `getResources()`.

Enfin, et surtout, vous pouvez récupérer un `Bitmap` avec `BitmapFactory.decodeStream(InputStream)`.

À l'opposé, au moment où l'on n'a plus besoin de `Bitmap`, on utilise dessus la méthode `void recycle()`. En effet, ça semble une habitude mais `Bitmap` n'est aussi qu'une interface et `recycle()` permet de libérer toutes les références à certains objets de manière à ce qu'ils puissent être ramassés par le *garbage collector*.

Après cette opération, le `Bitmap` n'est plus valide, vous ne pourrez plus l'utiliser ou faire d'opération dessus.

23.1.2. Le pinceau

Pour être tout à fait exact, `Paint` représente à la fois le pinceau et la palette. On peut créer un objet simplement sans passer de paramètre, mais il est possible d'être plus précis en indiquant des fanions. Par exemple, pour avoir des dessins plus nets (mais peut-être plus gourmands en ressources), on ajoute les fanions `Paint.ANTI_ALIAS_FLAG` et `Paint.DITHER_FLAG` :

```

```

La première chose est de déterminer ce qu'on veut dessiner : les contours d'une figure sans son intérieur, ou uniquement l'intérieur, ou bien même les contours *et* l'intérieur ? Afin d'assigner une valeur, on utilise `void setStyle(Paint.Style style)` :

```

```

On peut ensuite assigner une couleur avec `void setColor(int color)`. Comme vous pouvez le voir, cette méthode prend un entier, mais quelles valeurs peut-on lui donner ? Eh bien, pour vous aider dans cette tâche, Android fournit la classe `Color` qui va calculer pour vous la couleur en fonction de certains paramètres que vous passerez. Je pense particulièrement à `static int argb(int alpha, int red, int green, int blue)` qui dépend de la valeur de chaque composante (respectivement la transparence, le rouge, le vert et le bleu). On peut aussi penser à `static int parseColor(String colorString)` qui prend une chaîne de caractères comme on pourrait les trouver sur internet :

```

```

23.1.3. Le peintre

Enfin, on va pouvoir peindre ! Ici, rien de formidable, il existe surtout des méthodes qui expriment la forme à représenter. Tout d'abord, n'oubliez pas de donner un `Bitmap` au `Canvas`, sinon il n'aura pas de surface sur laquelle dessiner :

```

```

C'est tout ! Ensuite, pour dessiner une figure, il suffit d'appeler la méthode appropriée. Par exemple :

- `void drawColor(int color)` pour remplir la surface du `Bitmap` d'une couleur.
- `void drawRect(Rect r, Paint paint)` pour dessiner un rectangle.
- `void drawText(String text, float x, float y, Paint paint)` afin de dessiner… du texte. Eh oui, le texte se dessine aussi.

Vous trouverez plus de méthodes sur la page qui y est consacrée sur le site d'Android Developers.

23.2. Afficher notre toile

Il existe deux manières pour afficher nos œuvres d'art : sur n'importe quelle vue, ou sur une surface dédiée à cette tâche.

23.2.1. Sur une vue standard

Cette solution est la plus intéressante si votre surface de dessin n'a pas besoin d'être rapide ou fluide. C'est le cas quand on veut faire une banale vue personnalisée, mais pas quand on veut faire un jeu.

Il n'y a pas grand-chose à dire ici que vous ne sachiez déjà. Les dessins seront à effectuer dans la méthode de *callback* `void onDraw (Canvas canvas)` qui vous fournit le `Canvas` sur lequel dessiner. Ce `Canvas` contient déjà un `Bitmap` qui représente le dessin de la vue.

Cette méthode `onDraw(Canvas)` sera appelée à chaque fois que la vue juge que c'est nécessaire. Si vous voulez indiquer manuellement à la vue qu'elle doit se redessiner *le plus vite possible*, on peut le faire en utilisant la méthode `void invalidate()`.

Un appel à la méthode `invalidate()` n'est pas nécessairement instantané, il se peut qu'elle prenne un peu de temps puisque cet appel doit se faire dans le thread UI et passera par conséquent après toutes les actions en cours. Il est d'ailleurs possible d'invalider une vue depuis un autre thread avec la méthode `void postInvalidate()`.

23.2.2. Sur une surface dédiée à ce travail

Cette solution est déjà plus intéressante dès qu'il s'agit de faire un jeu, parce qu'elle permet de dessiner dans des threads différents du thread UI. Ainsi, au lieu d'avoir à attendre qu'Android déclare à notre vue qu'elle peut se redessiner, on aura notre propre thread dédié à cette tâche, donc sans encombrer le thread UI. Mais ce n'est pas tout ! En plus d'être plus rapide, cette surface peut être prise en charge par OpenGL si vous voulez effectuer des opérations graphiques encore plus compliquées.

Techniquement, la classe sur laquelle nous allons dessiner s'appelle `SurfaceView`. Cependant, nous n'allons pas la manipuler directement, nous allons passer par une couche d'abstraction représentée par la classe `SurfaceHolder`. Afin de récupérer un `SurfaceHolder` depuis un `SurfaceView`, il suffit d'appeler `SurfaceHolder getHolder()`. De plus, pour gérer correctement le cycle de vie de notre `SurfaceView`, on aura besoin d'implémenter l'interface `SurfaceHolder.Callback`, qui permet au `SurfaceView` de recevoir des informations sur les différentes phases et modifications qu'elle expérimente. Pour associer un `SurfaceView` à un `SurfaceHolder.Callback`, on utilise la méthode `void addCallback(SurfaceHolder.Callback callback)` sur le `SurfaceHolder` associé au `SurfaceView`. Cette opération doit être effectuée dès la création du `SurfaceView` afin de pouvoir prendre en compte son commencement.

N'ayez toujours qu'un thread au maximum qui manipule une `SurfaceView`, sinon gare aux soucis !

Ainsi, il nous faudra implémenter trois méthodes de *callback* qui réagiront à trois évènements différents :

- `void surfaceChanged(SurfaceHolder holder, int format, int width, int height)` sera enclenché à chaque fois que la surface est modifiée, c'est donc ici qu'on mettra à jour notre image. Mis à part certains paramètres que vous connaissez déjà tels que la largeur `width`, la hauteur `height` et le format `PixelFormat`, on trouve un nouvel objet de type `SurfaceHolder`. Un `SurfaceHolder` est une interface qui représente la surface sur laquelle dessiner. Mais ce n'est pas avec lui qu'on dessine, c'est bien avec un `Canvas`, de façon à ne pas manipuler directement le `SurfaceView`.
- `void surfaceCreated(SurfaceHolder holder)` sera quant à elle déclenchée uniquement à la création de la surface. On peut donc commencer à dessiner ici.
- Enfin, `void surfaceDestroyed(SurfaceHolder holder)` est déclenchée dès que la surface est détruite, de façon à ce que vous sachiez quand arrêter votre thread. Après que cette méthode a été appelée, la surface n'est plus disponible du tout.

Passons maintenant au dessin en tant que tel. Comme d'habitude, il faudra dessiner à l'aide d'un `Canvas`, sachant qu'il a déjà un `Bitmap` attribué. Comme notre dessin est dynamique, il faut d'abord bloquer le `Canvas`, c'est-à-dire immobiliser l'image actuelle pour pouvoir dessiner dessus. Pour bloquer le `Canvas`, il suffit d'utiliser la méthode `Canvas lockCanvas()`. Puis, une fois votre dessin terminé, vous pouvez le remettre en route avec `void unlockCanvasAndPost(Canvas canvas)`. C'est indispensable, sinon votre téléphone restera bloqué.

La surface sur laquelle se fait le dessin sera supprimée à chaque fois que l'activité se met en pause et sera recréée dès que l'activité reprendra, il faut donc interrompre le dessin à ces moments-là.

Pour économiser un peu notre processeur, on va instaurer une pause dans la boucle principale. En effet, si on ne fait pas de boucle, le thread va dessiner sans cesse le plus vite, alors que l'oeil humain ne sera pas capable de voir la majorité des images qui seront dessinées. C'est pourquoi nous allons rajouter un morceau de code qui impose au thread de ne plus calculer pendant 20 millisecondes. De cette manière, on affichera 50 images par seconde en moyenne, l'illusion sera parfaite pour l'utilisateur et la batterie de vos utilisateurs vous remercie déjà :

Voici un exemple d'implémentation de `SurfaceView` :

- On a besoin de plusieurs éléments pour dessiner : un `Canvas`, un `Bitmap` et un `Paint`.
- Le `Bitmap` est l'objet qui contiendra le dessin, on peut le comparer à la toile d'un tableau.

— Un `Paint` est tout simplement un objet qui représente un pinceau, on peut lui attribuer une couleur ou une épaisseur par exemple.
— Pour faire le lien entre une toile et un pinceau, on a besoin d'un peintre ! Ce peintre est un `Canvas`. Il contient un `Bitmap` et dessine dessus avec un `Paint`. Il est quand même assez rare qu'on fournisse un `Bitmap` à un `Canvas`, puisqu'en général la vue qui affichera le dessin nous fournira un `Canvas` qui possède déjà un `Bitmap` tout configuré.
— En tant que programmeur, vous êtes un client qui ordonne au peintre d'effectuer un dessin, ce qui fait que vous n'avez pas à vous préoccuper de manipuler le pinceau ou la toile, le peintre le fera pour vous.
— Pour afficher un dessin, on peut soit le faire avec une vue comme on l'a vu dans la seconde partie, soit créer carrément une surface dédiée au dessin. Cette solution est à privilégier quand on veut créer un jeu par exemple. Le plus gros point faible est qu'on doit utiliser des threads.

24. La localisation et les cartes

Nous sommes nombreux à avoir déjà utilisé Google Maps. Que ce soit pour trouver le vendeur de pizzas le plus proche, tracer l'itinéraire entre chez soi et le supermarché ou, encore mieux, regarder sa propre maison avec les images satellite.

Avec les progrès de la miniaturisation, la plupart — voire la quasi-totalité — des terminaux sont équipés de puces GPS. La géolocalisation est ainsi devenue un élément du quotidien qu'on retrouve dans énormément d'applications. On peut penser aux applications de navigation aidée par GPS, mais aussi aux applications sportives qui suivent nos efforts et élaborent des statistiques, ou encore aux applications pour noter les restaurants et les situer. On trouve ainsi deux API qui sont liées au concept de localisation :

— Une API qui permet de localiser l'appareil.
— Une API qui permet d'afficher des cartes.

24.1. La localisation

24.1.1. Préambule

On trouve tous les outils de localisation dans le package `android.location`.

Le GPS est la solution la plus efficace pour localiser un appareil, cependant il s'agit aussi de la plus coûteuse en batterie. Une autre solution courante est de se localiser à l'aide des points d'accès WiFi à proximité et de la distance mesurée avec les antennes relais du réseau mobile les plus proches (par triangulation).

Tout d'abord, vous devrez demander la permission dans le Manifest pour utiliser les fonctionnalités de localisation. Si vous voulez utiliser la géolocalisation (par GPS, donc), utilisez `ACCESS_FINE_LOCATION` ; pour une localisation plus imprécise par WiFi et antennes relais, utilisez `ACCESS_COARSE_LOCATION`. Enfin, si vous voulez utiliser les deux types de localisation, vous pouvez déclarer uniquement `ACCESS_FINE_LOCATION`, qui comprend toujours `ACCESS_COARSE_LOCATION` :

```
```

Ensuite, on va faire appel à un nouveau service système pour accéder à ces fonctionnalités : `LocationManager`, que l'on récupère de cette manière :

24.1.2. Les fournisseurs de position

Vous aurez ensuite besoin d'un fournisseur de position qui sera dans la capacité de déterminer la position actuelle. On a par un exemple un fournisseur pour le GPS et un autre pour les antennes relais. Ces fournisseurs dériveront de la classe abstraite `LocationProvider`. Il existe plusieurs méthodes pour récupérer les fournisseurs de position disponibles sur l'appareil. Pour récupérer le nom de tous les fournisseurs, il suffit de faire `List<String> getAllProviders()`. Le problème de cette méthode est qu'elle va récupérer tous les fournisseurs qui existent, même si l'application n'a pas le droit de les utiliser ou qu'ils sont désactivés par l'utilisateur.

Pour ne récupérer que le nom des fournisseurs qui sont réellement utilisables, on utilisera `List<String> getProviders(boolean enabledOnly)`. Enfin, on peut obtenir un `LocationProvider` à partir de son nom avec `LocationProvider getProvider(String name)` :

Les noms des fournisseurs sont contenus dans des constantes, comme `LocationManager.GPS_PROVIDER` pour le GPS et `LocationManager.NETWORK_PROVIDER` pour la triangulation.

Cependant, il se peut que vous ayez à sélectionner un fournisseur en fonction de critères bien précis. Pour cela, il vous faudra créer un objet de type `Criteria`. Par exemple, pour configurer tous les critères, on fera :

Pour obtenir tous les fournisseurs qui correspondent à ces critères, on utilise `List<String> getProviders(Criteria criteria, boolean enabledOnly)` et, pour obtenir le fournisseur qui correspond le plus, on utilise `String getBestProvider(Criteria criteria, boolean enabledOnly)`.

24.1.3. Obtenir des notifications du fournisseur

Pour obtenir la dernière position connue de l'appareil, utilisez `Location getLastKnownLocation(String provider)`.

La dernière position connue n'est pas forcément la position *actuelle* de l'appareil. En effet, il faut demander à mettre à jour la position pour que celle-ci soit renouvelée dans le fournisseur. Si vous voulez faire en sorte que le fournisseur se mette à jour automatiquement à une certaine période ou tous les `x` mètres, on peut utiliser la méthode `void requestLocationUpdates(String provider, long minTime, float minDistance, LocationListener listener)` avec :

— `provider` le fournisseur de position.
— `minTime` la période entre deux mises à jour en millisecondes. Il faut mettre une valeur supérieure à 0, sinon le fournisseur ne sera pas mis à jour périodiquement. D'ailleurs, ne mettez pas de valeur en dessous de 60 000 ms pour préserver la batterie.
— `minDistance` la période entre deux mises à jour en mètres. Tout comme pour `minTime`, il faut mettre une valeur supérieure à 0, sinon ce critère ne sera pas pris en compte. De plus, on privilégie quand même `minTime` parce qu'il consomme moins de batterie.
— `listener` est l'écouteur qui sera lancé dès que le fournisseur sera activé.

Ainsi, il faut que vous utilisiez l'interface `LocationListener`, dont la méthode `void onLocationChanged(Location location)` sera déclenchée à chaque mise à jour. Cette méthode de *callback* contient un objet de type `Location` duquel on peut extraire des informations sur l'emplacement donné. Par exemple, on peut récupérer la latitude avec `double getLatitude()` et la longitude avec `double getLongitude()` :

```

```

Cependant, ce code ne fonctionnera que si votre application est en cours de fonctionnement. Mais si vous souhaitez recevoir des notifications même quand l'application ne fonctionne pas ? On peut utiliser à la place `void requestLocationUpdates(String provider, long minTime, float minDistance, PendingIntent intent)` où le `PendingIntent` contenu dans `intent` sera lancé à chaque mise à jour du fournisseur. L'emplacement sera contenu dans un extra dont la clé est `KEY_LOCATION_CHANGED`.

```

```

On le recevra ensuite dans :

```

```

Enfin, vous pouvez désactiver les notifications avec `removeUpdates` en lui donnant le `LocationListener` ou le `PendingIntent` concerné. Si vous ne le faites pas, votre application continuera à recevoir des notifications après que tous les composants de l'application auront été fermés.

24.1.4. Les alertes de proximité

Dernière fonctionnalité que nous allons voir, le fait d'être informés quand on s'approche d'un endroit ou qu'on s'en éloigne. Cet endroit peut être symbolisé par un cercle dont on va préciser le centre et le rayon. Ainsi, si on entre dans ce cercle ou qu'on sort de ce cercle, l'alerte est lancée.

Le prototype de la méthode qui peut créer une alerte de proximité est `void addProximityAlert(double latitude, double longitude, float radius, long expiration, PendingIntent intent)` avec :

— La `latitude` et la `longitude` du centre du cercle.
— Le rayon d'effet est précisé dans `radius` en mètres.

— `expiration` permet de déclarer combien de temps cette alerte est valable. Tout nombre en dessous de 0 signifie qu'il n'y a pas d'expiration possible.
— Enfin, comme vous vous en doutez, on donne aussi le `PendingIntent` qui sera lancé quand cette alerte est déclenchée, avec `intent`.

Cette fois, l'intent contiendra un booléen en extra, dont la clé est `KEY_PROXIMITY_ENTERING` et la valeur sera `true` si on entre dans la zone et `false` si on en sort.

```

```

On le recevra ensuite dans :

```

```

Enfin, il faut désactiver une alerte de proximité avec `void removeProximityAlert(PendingIntent intent)`.

24.2. Afficher des cartes

C'est bien de pouvoir récupérer l'emplacement de l'appareil à l'aide du GPS, mais il faut avouer que si on ne peut pas l'afficher sur une carte c'est sacrément moins sympa ! Pour cela, on va passer par l'API Google Maps.

Contrairement aux API que nous avons vues pour l'instant, l'API pour Google Maps n'est pas intégrée à Android, mais appartient à une extension appelée « Google APIs », comme nous l'avons vu au cours des premiers chapitres. Ainsi, quand vous allez créer un projet, au lieu de sélectionner `Android 2.1 (API 7)`, on va sélectionner `Google APIs (Google Inc.) (API 7)`. Et si vous ne trouvez pas `Google APIs (Google Inc.) (API 7)`, c'est qu'il vous faudra le télécharger, auquel cas je vous renvoie au chapitre 2 de la première partie qui traite de ce sujet. Il faut ensuite l'ajouter dans le Manifest. Pour cela, on doit ajouter la ligne suivante dans le nœud `application` :

```

```

Cette opération rendra votre application invisible sur le Play Store si l'appareil n'a pas Google Maps. De plus, n'oubliez pas d'ajouter une permission pour accéder à internet, les cartes ne se téléchargent pas par magie :

```

```

24.2.1. Obtenir une clé pour utiliser Google Maps

Pour pouvoir utiliser Google Maps, il vous faudra demander l'autorisation pour accéder aux services sur internet. Pour cela, vous allez demander une clé .

Comme vous pouvez le voir, pour obtenir la clé, vous aurez besoin de l'empreinte MD5 du certificat que vous utilisez pour signer votre application. Si vous ne comprenez pas un traître mot de ce que je viens de dire, c'est que vous n'avez pas lu l'annexe sur la publication d'applications , ce que je vous invite à faire, en particulier la partie sur la signature.

Ensuite, la première chose à faire est de repérer où se trouve votre certificat. Si vous travaillez en mode debug, alors Eclipse va générer un certificat pour vous :

— Sous Windows, le certificat par défaut se trouve dans le répertoire consacré à votre compte utilisateur dans `\.android\debug.keystore`. Si vous avez Windows Vista, 7 ou 8, alors ce répertoire utilisateur se trouve par défaut dans `C:\Users\nom_d_utilisateur`. Pour Windows XP, il se trouve dans `C:\Documents and Settings\nom_d_utilisateur`.
— Pour Mac OS et Linux, il se trouve dans `~/.android/`.

Puis il vous suffira de lancer la commande suivante dans un terminal :

```
```

Deux choses auxquelles vous devez faire attention :

— Si cette commande ne marche pas, c'est que vous avez mal configuré votre `PATH` au moment de l'installation de Java. Ce n'est pas grave, il vous suffit d'aller à l'emplacement où se trouve l'outil *keytool* et d'effectuer la commande. Chez moi, il s'agit de `C:\Program Files (x86)\Java\jre7\bin`.
— Et si comme moi vous utilisez Java 7 au lieu de Java 6, vous devrez rajouter `-v` à la commande, ce qui donne `keytool -list -keystore "Emplacement du certificat" -storepass android -keypass android -v`

Ainsi, ce que j'ai fait pour obtenir ma clé MD5, c'est :

— Aller dans `C:\Program Files (x86)\Java\jre7\bin` ;
— Taper `keytool -list -keystore "C:\Users\Apollidore\.android\debug.keystore" -storepass android -keypass android -v` ;
— Repérer la ligne où il était écrit « MD5 », comme à la figure suivante.

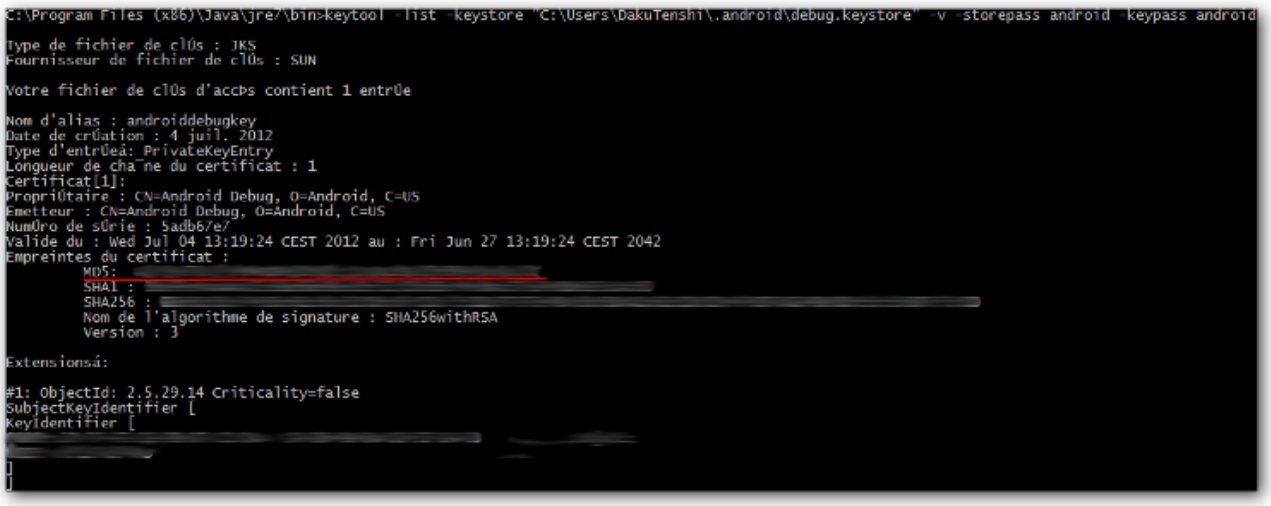

FIGURE 24.1. – Comme vous pouvez le voir, on évite de montrer à tout le monde les codes obtenus, sinon on pourrait se faire passer pour vous

Insérez ensuite le code MD5 sur le site pour obtenir une clé que vous pourrez utiliser dans l'application, mais nous verrons comment procéder plus tard.

24.2.2. L'activité qui contiendra la carte

Tout d'abord, pour simplifier l'utilisation des cartes, chaque activité qui contiendra une carte sera de type `MapActivity`. Vous aurez besoin d'implémenter au moins deux méthodes : `onCreate(Bundle)` et `protected boolean isRouteDisplayed()`. Cette méthode permet de savoir si d'une manière ou d'une autre la vue qui affichera la carte permettra de visualiser des informations de type itinéraire ou parcours. Renvoyez `true` si c'est le cas, sinon renvoyez `false`. Enfin, vous devez aussi implémenter la méthode `protected boolean isLocationDisplayed()` qui renverra `true` si votre application affiche l'emplacement actuel de l'utilisateur.

Vous devez implémenter ces deux méthodes pour utiliser légalement cette API.

24.2.3. La carte en elle-même

Pour gérer l'affichage de la carte, on va utiliser une `MapView` qui se trouve elle aussi dans le package `com.google.android.maps`. Au moment de la déclaration en XML, on va penser surtout à deux attributs :

— `android:clickable`, si vous voulez que l'utilisateur puisse cliquer sur la carte ou se déplacer dans la carte ;
— `android:apiKey`, pour préciser la clé que vous avez récupérée sur le site.

Par exemple :

Ne mettez pas plus d'une `MapActivity` et une `MapView` par application, ce n'est pas très bien supporté pour l'instant, elles pourraient entrer en conflit.

Comme vous le savez probablement, il existe trois modes d'affichage sur une carte Google Maps :

- Si vous voulez afficher la vue satellitaire, utilisez `mapView.setSatellite(true)`.
- Si vous voulez afficher les routes, les noms des rues et tout ce qu'on peut attendre d'une carte routière, utilisez `mapView.setTraffic(true)`.
- Et si vous voulez afficher la possibilité de passer en mode Street View (vous savez, ce mode qui prend le point de vue d'un piéton au milieu d'une rue), alors utilisez `mapView.setStreetView(true)`. Ce mode n'est pas compatible avec le mode trafic.

24.2.4. Le contrôleur

Votre carte affiche ce que vous voulez… enfin presque. Si vous voulez faire un zoom ou déplacer la région actuellement visualisée, il vous faudra utiliser un `MapController`. Pour récupérer le `MapController` associé à une `MapView`, il suffit de faire `MapController getController()`.

24.2.4.1. Le zoom

Il existe 21 niveaux de zoom, et chacun d'eux représente un doublement dans le nombre de pixels qu'affiche une même zone. Ainsi en zoom 1, on peut voir toute la planète (plusieurs fois), alors qu'en zoom 21 on peut voir le chien du voisin. Pour contrôler le zoom, utilisez sur le contrôleur la méthode `int setZoom(int zoomLevel)` qui retourne le nouveau niveau de zoom. Vous pouvez zoomer et dézoomer d'un niveau avec respectivement `boolean zoomIn()` et `boolean zoomOut()`.

24.2.4.2. Se déplacer dans la carte

Pour modifier l'emplacement qu'affiche le centre de la carte, il faut utiliser la méthode `void setCenter(GeoPoint point)` (ou `public void animateTo(GeoPoint point)` si vous voulez une animation). Comme vous pouvez le voir, ces deux méthodes prennent des objets de type `GeoPoint`. Très simplement, un `GeoPoint` est utilisé pour représenter un emplacement sur la planète, en lui donnant une longitude et une latitude. Ainsi, le `GeoPoint` qui représente le bâtiment où se situe Simple IT sera créé de cette manière :

Ce qui est pratique, c'est que ce calque permet d'effectuer une action dès que le GPS détecte la position de l'utilisateur avec la méthode `boolean runOnFirstFix(Runnable runnable)`, par exemple pour zoomer sur la position de l'utilisateur à ce moment-là :

```
```

24.2.5. Utiliser les calques pour afficher des informations complémentaires

Parfois, afficher une carte n'est pas suffisant, on veut en plus y ajouter des informations. Et si je veux afficher Zozor, la mascotte du Site du Zéro, sur ma carte à l'emplacement où se trouve Simple IT ? Et si en plus je voulais détecter les clics des utilisateurs sur un emplacement de la carte ? Il est possible de rajouter plusieurs couches sur lesquelles dessiner et qui sauront réagir aux évènements communs. Ces couches sont des calques, qui sont des objets de type `Overlay`.

24.2.5.1. Ajouter des calques

Pour récupérer la liste des calques que contient la carte, on fait `List<Overlay> getOverlays()`. Il suffit d'ajouter des `Overlay` à cette liste pour qu'ils soient dessinés sur la carte. Vous pouvez très bien accumuler les calques sur une carte, de manière à dessiner des choses les unes sur les autres. Chaque calque ajouté se superpose aux précédents, il se place au-dessus. Ainsi, les dessins de ce nouveau calque se placeront au-dessus des précédents et les évènements, par exemple les touchers, seront gérés de la manière suivante : le calque qui se trouve au sommet recevra en premier l'évènement. Si ce calque n'est pas capable de gérer cet évènement, ce dernier est alors transmis aux calques qui se trouvent en dessous. Néanmoins, si le calque est effectivement capable de gérer cet évènement, alors il s'en charge et l'évènement ne sera plus propagé.

Enfin, pour indiquer qu'on a rajouté un `Overlay`, on utilise la méthode `void postInvalidate()` pour faire se redessiner la `MapView` :

```
```

24.2.5.2. Dessiner sur un calque

Cependant, si vous voulez ajouter un point d'intérêt — on dit aussi un « POI » —, c'est-à-dire un endroit remarquable sur la carte, je vous recommande plutôt d'utiliser la classe `ItemizedOverlay`.

On implémente en général au moins deux méthodes. La première est `void draw(Canvas canvas, MapView mapView, boolean shadow)` qui décrit le dessin à effectuer. On dessine sur le `canvas` et on projette le dessin du `Canvas` sur la vue `mapView`. En ce qui concerne `shadow`, c'est plus compliqué. En fait, cette méthode `draw` est appelée deux fois : une première fois où `shadow` vaut `false` pour dessiner normalement et une seconde où `shadow` vaut `true` pour rajouter des ombres à votre dessin.

On a un problème d'interface entre le `Canvas` et la `MapView` : les coordonnées sur le `Canvas` sont en `Point` et les coordonnées sur la `MapView` sont en `GeoPoint`, il nous faut donc un moyen pour convertir nos `Point` en `GeoPoint`. Pour cela, on utilise une `Projection` avec la méthode `Projection getProjection()` sur la `MapView`. Pour obtenir un `GeoPoint` depuis un `Point`, on peut faire `GeoPoint fromPixels(int x, int y)` et, pour obtenir un `Point` depuis un `GeoPoint`, on peut faire `Point toPixels(GeoPoint in, Point out)`. Par exemple :

```
```

Ainsi, pour dessiner un point rouge à l'emplacement du bâtiment de Simple IT :

```
```

24.2.5.3. Gérer les évènements sur un calque

La méthode de *callback* qui sera appelée quand l'utilisateur appuiera sur le calque s'appelle `boolean onTap(GeoPoint p, MapView mapView)` avec `p` l'endroit où l'utilisateur a appuyé et `mapView` la carte sur laquelle il a appuyé. Il vous est demandé de renvoyer `true` si l'évènement a été géré (auquel cas il ne sera plus transmis aux couches qui se trouvent en dessous).

```
```

24.2.6. Quelques calques spécifiques

Maintenant que vous pouvez créer tous les calques que vous voulez, nous allons en voir quelques-uns qui permettent de nous faciliter grandement la vie dès qu'il s'agit de faire quelques tâches standards.

24.2.6.1. Afficher la position actuelle

Les calques de type `MyLocationOverlay` permettent d'afficher votre position actuelle ainsi que votre orientation à l'aide d'un capteur qui est disponible dans la plupart des appareils de nos jours. Pour activer l'affichage de la position actuelle, il suffit d'utiliser `boolean enableMyLocation()` et, pour afficher l'orientation, il suffit d'utiliser `boolean enableCompass()`. Afin d'économiser la batterie, il est conseillé de désactiver ces deux fonctionnalités quand l'activité passe en pause, puis de les réactiver après :

```
```

24.2.6.2. Ajouter des marqueurs

Pour marquer l'endroit où se trouvait le bâtiment de Simple IT, nous avons ajouté un rond rouge sur la carte, cependant il existe un type d'objets qui permet de faire ce genre de tâches très facilement. Il s'agit d'`OverlayItem`. Vous aurez aussi besoin d'`ItemizedOverlay` qui est une liste d'`OverlayItem` à gérer et c'est elle qui fera les dessins, la gestion des évènements, etc.

Nous allons donc créer une classe qui dérive d'`ItemizedOverlay<OverlayItem>`. Nous aurons ensuite à nous occuper du constructeur. Dans celui-ci, il faudra passer un `Drawable` qui représentera le marqueur visuel de nos points d'intérêt. De plus, dans le constructeur, il vous faudra construire les différents `OverlayItem` et déclarer quand vous aurez fini avec la méthode `void populate()` :

```

```

Vous remarquerez qu'un `OverlayItem` prend trois paramètres :

— Le premier est le `GeoPoint` où se trouve l'objet sur la carte.
— Le deuxième est le titre à attribuer au point d'intérêt.
— Le dernier est un court texte descriptif.

Enfin, il existe deux autres méthodes que vous devez implémenter :

— `int size()`, qui retourne le nombre de points d'intérêt dans votre liste.
— `Item createItem(int i)`, pour retourner le i-ième élément de votre liste.

— Il existe plusieurs manières de détecter la position d'un appareil. La plus efficace est la méthode qui exploite la puce GPS, mais c'est aussi la plus consommatrice en énergie. La seconde méthode se base plutôt sur les réseaux Wifi et les capteurs internes du téléphone. Elle est beaucoup moins précise mais peut être utilisée en dernier recours si l'utilisateur a désactivé l'utilisation de la puce GPS.
— Il est possible de récupérer la position de l'utilisateur avec le `LocationManager`. Il est aussi possible de créer des évènements qui permettent de réagir au statut de l'utilisateur avec `LocationListener`.
— Il est possible d'afficher des cartes fournies par Google Maps avec l'API Google Maps. Ainsi, on gérera une carte avec une `MapActivity` et on l'affichera dans une `MapView`.
— Le plus intéressant est de pouvoir ajouter des calques, afin de fournir des informations basées sur la géographie à l'utilisateur. On peut par exemple proposer à un utilisateur d'afficher les restaurants qui se trouvent à proximité. Un calque est un objet de type `Overlay`.

25. La téléphonie

Il y a de grandes chances pour que votre appareil sous Android soit un téléphone. Et comme tous les téléphones, il est capable d'appeler ou d'envoyer des messages. Et comme nous sommes sous Android, il est possible de supplanter ces fonctionnalités natives pour les gérer nous-mêmes.

Encore une fois, tout le monde n'aura pas besoin de ce dont on va parler. Mais il peut très bien arriver que vous ayez envie qu'appuyer sur un bouton appelle un numéro d'urgence, ou le numéro d'un médecin, ou quoi que ce soit.

De plus, il faut savoir que le SMS — vous savez les petits messages courts qui font 160 caractères au maximum — est le moyen le plus courant pour communiquer entre deux appareils mobiles, il est donc très courant qu'un utilisateur ait envie d'en envoyer un à un instant `t`. Même s'ils sont beaucoup moins utilisés, les MMS — comme un SMS, mais avec un média (son, vidéo ou image) qui l'accompagne — sont monnaie courante.

25.1. Téléphoner

La première chose qu'on va faire, c'est s'assurer que l'appareil sur lequel fonctionnera votre application peut téléphoner, sinon notre application de téléphonie n'aura absolument aucun sens. Pour cela, on va indiquer dans notre Manifest que l'application ne peut marcher sans la téléphonie, en lui ajoutant la ligne suivante :

```
```

De cette manière, les utilisateurs ne pourront pas télécharger votre application sur le Play Store s'ils se trouvent sur leur tablette par exemple.

Maintenant, d'un point de vue technique, nous allons utiliser l'API téléphonique qui est incarnée par la classe `TelephonyManager`. Les méthodes fournies dans cette classe permettent d'obtenir des informations sur le réseau et d'accéder à des informations sur l'abonné. Vous l'aurez remarqué, il s'agit encore une fois d'un `Manager` ; ainsi, pour l'obtenir, on doit en demander l'accès au `Context` :

```
```

25.1.1. Obtenir des informations

Ensuite, si nous voulons obtenir des informations sur l'appareil, il faut demander la permission :

25.1.1.1. Informations statiques

Tout d'abord, on peut déterminer le type du téléphone avec `int getPhoneType()`. Cette méthode peut retourner trois valeurs :

- `TelephonyManager.PHONE_TYPE_NONE` si l'appareil n'est pas un téléphone ou ne peut pas téléphoner.
- `TelephonyManager.PHONE_TYPE_GSM` si le téléphone exploite la norme de téléphonie mobile GSM. En France, ce sera le cas tout le temps.
- `TelephonyManager.PHONE_TYPE_CDMA` si le téléphone exploite la norme de téléphonie mobile CDMA. C'est une technologie vieillissante, mais encore très en vogue en Amérique du Nord.

Pour obtenir un identifiant unique de l'appareil, vous pouvez utiliser `String getDeviceId()`, et il est (parfois) possible d'obtenir le numéro de téléphone de l'utilisateur avec `String getLineNumber()`.

25.1.1.2. Informations dynamiques

Les informations précédentes étaient statiques, il y avait très peu de risques qu'elles évoluent pendant la durée de l'exécution de l'application. Cependant, il existe des données liées au réseau qui risquent de changer de manière régulière. Pour observer ces changements, on va passer par une interface dédiée : `PhoneStateListener`. On peut ensuite indiquer quels changements d'état on veut écouter avec le `TelephonyManager` en utilisant la méthode `void listen (PhoneStateListener listener, int events)` avec `events` des flags pour indiquer quels évènements on veut écouter. On note par exemple la présence des flags suivants :

- `PhoneStateListener.LISTEN_CALL_STATE` pour savoir que l'appareil déclenche ou reçoit un appel.
- `PhoneStateListener.LISTEN_DATA_CONNECTION_STATE` pour l'état de la connexion avec internet.
- `PhoneStateListener.LISTEN_DATA_ACTIVITY` pour l'état de l'échange des données avec internet.
- `PhoneStateListener.LISTEN_CELL_LOCATION` pour être notifié des déplacements de l'appareil.

Bien entendu, si vous voulez que l'appareil puisse écouter les déplacements, il vous faudra demander la permission avec `ACCESS_COARSE_LOCATION`, comme pour la localisation expliquée au chapitre précédent.

Ensuite, à chaque évènement correspond une méthode de *callback* à définir dans votre implémentation de `PhoneStateListener` :

```
```

25.1.2. Téléphoner

Pour téléphoner, c'est très simple. En fait, vous savez déjà le faire. Vous l'avez peut-être même déjà fait. Il vous suffit de lancer un `Intent` qui a pour action `Intent.ACTION_CALL` et pour données `tel:numéro_de_téléphone` :

```
```

Cette action va lancer l'activité du combiné téléphonique pour que l'utilisateur puisse initier l'appel par lui-même. Ainsi, il n'y a pas besoin d'autorisation puisqu'au final l'utilisateur doit amorcer l'appel manuellement. Cependant, il se peut que vous souhaitiez que votre application lance l'appel directement, auquel cas vous devrez demander une permission particulière :

```
```

25.2. Envoyer et recevoir des SMS et MMS

25.2.1. L'envoi

Tout comme pour passer des appels, il existe deux manières de faire : soit avec l'application liée aux SMS, soit directement par l'application.

25.2.1.1. Prise en charge par une autre application

Pour transmettre un SMS à une application qui sera en charge de l'envoyer, il suffit d'utiliser un `Intent`. Il utilisera comme action `Intent.ACTION_SENDTO`, aura pour données un `smsto:numéro_de_téléphone` pour indiquer à qui sera envoyé le SMS, et enfin aura un extra de titre `sms_body` qui indiquera le contenu du message :

```
```

Pour faire de même avec un MMS, c'est déjà plus compliqué. Déjà, les MMS ne fonctionnent pas avec `SENDTO` mais avec `SEND` tout court. De plus, le numéro de téléphone de destination devra être défini dans un extra qui s'appellera `address`. Enfin, le média associé au MMS sera ajouté à l'intent dans les données et dans un extra de nom `Intent.EXTRA_STREAM` à l'aide d'une URI qui pointe vers lui :

25.2.1.2. Prise en charge directe

Tout d'abord, on a besoin de demander la permission :

Pour envoyer directement un SMS sans passer par une application externe, on utilise la classe `SmsManager`.

? Ah ! J'imagine qu'on peut la récupérer en faisant `Context.getSystemService(Context.SMS_SERVICE)`, j'ai compris le truc maintenant !

Pour une fois, même si le nom de la classe se termine par « Manager », on va instancier un objet avec la méthode `static SmsManager SmsManager.getDefault()`. Pour envoyer un message, il suffit d'utiliser la méthode `void sendTextMessage(String destinationAddress, String scAddress, String text, PendingIntent sentIntent, PendingIntent deliveryIntent)` :

- Il vous faut écrire le numéro du destinataire dans `destinationAddress`.
- Vous pouvez spécifier un centre de service d'adressage qui va gérer l'envoi du SMS dans `scAddress`. Si vous n'avez aucune idée de quoi mettre, un `null` sera suffisant.
- `text` contient le texte à envoyer.
- Il vous est possible d'insérer un `PendingIntent` dans `sentIntent` si vous souhaitez avoir des nouvelles de la transmission du message. Ce `PendingIntent` sera transmis à tout le système de façon à ce que vous puissiez le récupérer si vous le voulez. Le `PendingIntent` contiendra le code de résultat `Activity.RESULT_OK` si tout s'est bien passé. Enfin, vous pouvez aussi très bien mettre `null`.
- Encore une fois, vous pouvez mettre un `PendingIntent` dans `deliveryIntent` si vous souhaitez avoir des informations complémentaires sur la transmission.

On peut ainsi envisager un exemple simple :

La taille maximum d'un SMS est de 160 caractères ! Vous pouvez cependant couper un message trop long avec `ArrayList<String> divideMessage(String text)`, puis vous pouvez envoyer les messages de façon à ce qu'ils soient liés les uns aux autres avec `void sendMultipartTextMessage(String destinationAddress, String scAddress, ArrayList<String> parts, ArrayList<PendingIntent> sentIntents, ArrayList<PendingIntent> deliveryIntents)`, les paramètres étant analogues à ceux de la méthode précédente.

Malheureusement, envoyer des MMS directement n'est pas aussi simple, c'est même tellement complexe que je ne l'aborderai pas !

25.2.1.3. Recevoir des SMS

On va faire ici quelque chose d'un peu étrange. Disons le carrément, on va s'enfoncer dans la quatrième dimension. En fait, recevoir des SMS n'est pas réellement prévu de manière officielle dans le SDK. C'est à vous de voir si vous voulez le faire.

La première chose à faire est de demander la permission dans le Manifest :

```
```

Ensuite, dès que le système reçoit un nouveau SMS, un broadcast intent est émis avec comme action `android.provider.Telephony.SMS_RECEIVED`. C'est donc à vous de développer un broadcast receiver qui gérera la réception du message. L'`Intent` qui enclenchera le `Receiver` contiendra un tableau d'objets qui s'appelle « pdus » dans les extras. Les PDU sont les données qui sont transmises et qui représentent le message, ou les messages s'il a été divisé en plusieurs. Vous pourrez ensuite, à partir de ce tableau d'objets, créer un tableau de `SmsMessage` avec la méthode `static SmsMessage SmsMessage.createFromPdu(byte[] pdu)` :

```
```

Vous pouvez ensuite récupérer des informations sur le message avec diverses méthodes : le contenu du message avec `String getMessageBody()`, le numéro de l'expéditeur avec `String getOriginatingAddress()` et le moment de l'envoi avec `long getTimestampMillis()`.

— On peut obtenir beaucoup d'informations différentes sur le téléphone et le réseau de téléphonie auquel il est relié à l'aide de `TelephonyManager`. Il est même possible de surveiller l'état d'un téléphone avec `PhoneStateListener` de manière à pouvoir réagir rapidement à ses changements d'état.
— Envoyer des SMS se fait aussi assez facilement grâce à `SmsMessage` ; en revanche, envoyer des MMS est beaucoup plus complexe et demande un travail important.

26. Le multimédia

Il y a une époque pas si lointaine où, quand on voulait écouter de la musique en faisant son jogging, il fallait avoir un lecteur dédié, un *walkman*. Et si on voulait regarder un film dans le train, il fallait un lecteur DVD portable. Heureusement, avec les progrès de la miniaturisation, il est maintenant possible de le faire n'importe où et n'importe quand, avec n'importe quel *smartphone*. Clairement, les appareils mobiles doivent désormais remplir de nouvelles fonctions, et il faut des applications pour assumer ces fonctions.

C'est pourquoi nous verrons ici comment lire des musiques ou des vidéos, qu'elles soient sur un support ou en *streaming*. Mais nous allons aussi voir comment effectuer des enregistrements audio et vidéo.

26.1. Le lecteur multimédia

26.1.1. Où trouver des fichiers multimédia ?

Il existe trois emplacements à partir desquels vous pourrez lire des fichiers multimédia :

1. Vous pouvez tout d'abord les insérer en tant que ressources dans votre projet, auquel cas il faut les mettre dans le répertoire `res/raw`. Vous pouvez aussi les insérer dans le répertoire `assets/` afin d'y accéder avec une URI de type `file://android_asset/nom_du_fichier.format_du_fichier`. Il s'agit de la solution la plus simple, mais aussi de la moins souple.
2. Vous pouvez stocker les fichiers sur l'appareil, par exemple sur le répertoire local de l'application en interne, auquel cas ils ne seront disponibles que pour cette application, ou alors sur un support externe (genre carte SD), auquel cas ils seront disponibles pour toutes les applications de l'appareil.
3. Il est aussi possible de lire des fichiers en *streaming* sur internet.

26.1.2. Formats des fichiers qui peuvent être lus

Tout d'abord, pour le streaming, on accepte le RTSP, RTP et le streaming via HTTP.

Ensuite, je vais vous présenter tous les formats que connaît Android de base. En effet, il se peut que le constructeur de votre téléphone ait rajouté des capacités que je ne peux connaître. Ainsi, Android pourra toujours lire tous les fichiers présentés ci-dessous. Vous devriez comprendre toutes les colonnes de ce tableau, à l'exception peut-être de la colonne « **Encodeur** » : elle vous indique si oui ou non Android est capable de convertir un fichier vers ce format.

26.1.2.1. Audio

Format	Encodeur	Extension
AAC LC	**oui**	**3GPP (.3gp), MPEG-4 (.mp4, .m4a)**
HE-AACv1 (AAC+)		3GPP (.3gp), MPEG-4 (.mp4, .m4a)
HE-AACv2 (enhanced AAC+)		3GPP (.3gp), MPEG-4 (.mp4, .m4a)
AMR-NB	oui	3GPP (.3gp)
AMR-WB	oui	3GPP (.3gp)
MP3		MP3 (.mp3)
MIDI		Type 0 and 1 (.mid, .xmf, .mxmf), RTTTL/RTX (.rtttl, .rtx), OTA (.ota), iMelody (.imy)
Vorbis		Ogg (.ogg), Matroska (.mkv, Android 4.0+)
PCM/WAVE		WAVE (.wav)

26.1.2.2. Vidéo

Format	Encodeur	Extension
H.263	**oui**	**3GPP (.3gp), MPEG-4 (.mp4)**
H.264 AVC		3GPP (.3gp), MPEG-4 (.mp4)
MPEG-4 SP		3GPP (.3gp)

26.1.3. Le lecteur multimédia

26.1.3.1. Permissions

La première chose qu'on va faire, c'est penser aux permissions qu'il faut demander. Il n'y a pas de permission en particulier pour la lecture ou l'enregistrement ; en revanche, certaines fonctionnalités nécessitent quand même une autorisation. Par exemple, pour le streaming, il faut demander l'autorisation d'accéder à internet :

De même, il est possible que vous vouliez faire en sorte que l'appareil ne se mette jamais en veille de façon à ce que l'utilisateur puisse continuer à regarder une vidéo qui dure longtemps sans être interrompu :

```
```

26.1.3.2. La lecture

La lecture de fichiers multimédia se fait avec la classe `MediaPlayer`. Sa vie peut être représentée par une machine à état, c'est-à-dire qu'elle traverse différents états et que la transition entre chaque état est symbolisée par des appels à des méthodes.

? Comme pour une activité ?

Mais oui, exactement, vous avez tout compris !

Je pourrais très bien expliquer toutes les étapes et toutes les transitions, mais je doute que cela puisse vous être réellement utile, je ne ferais que vous embrouiller, je vais donc simplifier le processus. On va ainsi ne considérer que cinq états : **initialisé** quand on crée le lecteur, **préparé** quand on lui attribue un média, **démarré** tant que le média est joué, **en pause** quand la lecture est mise en pause ou **arrêté** quand elle est arrêtée, et enfin **terminé** quand la lecture est terminée.

Tout d'abord, pour créer un `MediaPlayer`, il existe un constructeur par défaut qui ne prend pas de paramètre. Un lecteur ainsi créé se trouve dans l'état **initialisé**. Vous pouvez ensuite lui indiquer un fichier à lire avec `void setDataSource(String path)` ou `void setDataSource(Context context, Uri uri)`. Il nous faut ensuite passer de l'état **initialisé** à **préparé** (c'est-à-dire que le lecteur aura commencé à lire le fichier dans sa mémoire pour pouvoir commencer la lecture). Pour cela, on utilise une méthode qui s'appelle simplement `void prepare()`.

! Cette méthode est synchrone, elle risque donc de bloquer le thread dans lequel elle se trouve. Ainsi, si vous appelez cette méthode dans le thread UI, vous risquez de le bloquer. En général, si vous essayez de lire dans un fichier cela devrait passer, mais pour un flux streaming il ne faut jamais faire cela. De manière générale, il faut appeler `prepare()` dans un thread différent du thread UI. Vous pouvez aussi appeler la méthode `void prepareAsync()`, qui est asynchrone et qui le fait de manière automatique pour vous.

Il est aussi possible de créer un lecteur multimédia directement préparé avec une méthode de type `create` :

```
```

Maintenant que notre lecteur est en mode **préparé**, on veut passer en mode **démarré** qui symbolise la lecture du média ! Pour passer en mode **démarré**, on utilise la méthode `void start()`.

On peut ensuite passer à deux états différents :

— L'état **en pause**, en utilisant la méthode `void pause()`. On peut revenir à tout moment à l'état **démarré** avec la méthode `void resume()`.
— L'état **arrêté**, qui est enclenché en utilisant la méthode `void stop()`. À partir de cet état, on ne peut pas revenir directement à **démarré**. En effet, il faudra repasser à l'état **préparé**, puis indiquer qu'on veut retourner au début du média (avec la méthode `void seekTo(int msec)` qui permet de se balader dans le média).

```

```

Enfin, une fois la lecture terminée, on passe à l'état **terminé**. À partir de là, on peut recommencer la lecture depuis le début avec `void start()`.

Enfin, n'oubliez pas de libérer la mémoire de votre lecteur multimédia avec la méthode `void release()`, on pourrait ainsi voir dans l'activité qui contient votre lecteur :

```

```

26.1.3.3. Le volume et l'avancement

Pour changer le volume du lecteur, il suffit d'utiliser la méthode `void setVolume(float leftVolume, float rightVolume)` avec `leftVolume` un entier entre 0.0f (pour silencieux) et 1.0f (pour le volume maximum) du côté gauche, et `rightVolume` idem pour le côté droit. De base, si vous appuyez sur les boutons pour changer le volume, seul le volume de la sonnerie sera modifié. Si vous voulez que ce soit le volume du lecteur qui change et non celui de la sonnerie, indiquez-le avec `void setVolumeControlStream(AudioManager.STREAM_MUSIC)`.

Si vous voulez que l'écran ne s'éteigne pas quand vous lisez un média, utilisez `void setScreenOnWhilePlaying(boolean screenOn)`.

Enfin, si vous voulez que la lecture se fasse en boucle, c'est-à-dire qu'une fois arrivé à **terminé** on passe à **démarré**, utilisez `void setLooping(boolean looping)`.

26.1.4. La lecture de vidéos

Maintenant qu'on sait lire des fichiers audio, on va faire en sorte de pouvoir regarder des vidéos. Eh oui, parce qu'en plus du son, on aura besoin de la vidéo. Pour cela, on aura besoin d'une vue qui s'appelle `VideoView`. Elle ne prend pas d'attributs particuliers en XML :

```

```

Puis on va attribuer à ce `VideoView` un `MediaController`. Mais qu'est-ce qu'un `MediaController` ? Nous n'en avons pas encore parlé ! Il s'agit en fait d'un layout qui permet de contrôler un média, aussi bien un son qu'une vidéo. Contrairement aux vues standards, on n'implémente pas un `MediaController` en XML mais dans le code. Tout d'abord, on va le construire avec `public MediaController(Context context)`, puis on l'attribue au `VideoView` avec `void setMediaController(MediaController controller)` :

```

```

26.2. Enregistrement

On aura besoin d'une permission pour enregistrer :

```

```

Il existe deux manières d'enregistrer.

26.2.1. Enregistrement sonore standard

Vous aurez besoin d'utiliser un `MediaRecorder` pour tous les enregistrements, dont les vidéos — mais nous le verrons plus tard. Ensuite c'est très simple, il suffit d'utiliser les méthodes suivantes :

- On indique quel est le matériel qui va enregistrer le son avec `void setAudioSource(int audio_source)`. Pour le micro, on lui donnera comme valeur `MediaRecorder.AudioSource.MIC`.
- Ensuite, vous pouvez choisir le format de sortie avec `void setOutputFormat(int output_format)`. De manière générale, on va mettre la valeur `MediaRecorder.OutputFormat.DEFAULT`, mais la valeur `MediaRecorder.OutputFormat.THREE_GPP` est aussi acceptable.
- Nous allons ensuite déclarer quelle méthode d'encodage audio nous voulons grâce à `void setAudioEncoder(int audio_encoder)`, qui prendra la plupart du temps `MediaRecorder.AudioEncoder.DEFAULT`.
- La prochaine chose à faire est de définir où sera enregistré le fichier avec `void setOutputFile(String path)`.
- Puis, comme pour le lecteur multimédia, on passe l'enregistreur en état **préparé** avec `void prepare()`.
- Enfin, on commence l'enregistrement avec `void start()`.

Pas facile à retenir, tout ça ! L'avantage ici, c'est que tout est automatique, alors vous n'avez « que » ces étapes à respecter.

```

```

Une fois que vous avez décidé de finir l'enregistrement, il vous suffit d'appeler la méthode `void stop()`, puis de libérer la mémoire :

26.2.2. Enregistrer du son au format brut

L'avantage du son au format brut, c'est qu'il n'est pas traité et permet par conséquent certains traitements que la méthode précédente ne permettait pas. De cette manière, le son est de bien meilleure qualité. On va ici gérer un flux sonore, et non des fichiers. C'est très pratique dès qu'il faut effectuer des analyses du signal en temps réel.

Nous allons utiliser ici un *buffer*, c'est-à-dire un emplacement mémoire temporaire qui fait l'intermédiaire entre deux matériels ou processus différents. Ici, le buffer récupérera les données du flux sonore pour que nous puissions les utiliser dans notre code.

La classe à utiliser cette fois est `AudioRecord`, et on peut en construire une instance avec `public AudioRecord(int audioSource, int sampleRateInHz, int channelConfig, int audioFormat, int bufferSizeInBytes)` où :

- `audioSource` est la source d'enregistrement ; souvent on utilisera le micro `MediaRecorder.AudioSource.MIC`.
- Le taux d'échantillonnage est à indiquer dans `sampleRateInHz`, même si dans la pratique on ne met que 44100.
- Il faut mettre dans `channelConfig` la configuration des canaux audio ; s'il s'agit de mono, on utilise `AudioFormat.CHANNEL_IN_MONO` ; s'il s'agit de stéréo, on utilise `AudioFormat.CHANNEL_IN_STEREO`.
- On peut préciser le format avec `audioFormat`, mais en pratique on mettra toujours `AudioFormat.ENCODING_PCM_16BIT`.
- Enfin,on va mettre la taille totale du buffer dans `bufferSizeInBytes`. Si vous n'y comprenez rien, ce n'est pas grave, la méthode `static int AudioRecord.getMinBufferSize(int sampleRateInHz, int channelConfig, int audioFormat)` vous fournira une bonne valeur à utiliser.

Une utilisation typique pourrait être :

Chaque lecture que nous ferons dans `AudioRecord` prendra la taille du buffer, il nous faudra donc avoir un tableau qui fait la taille de ce buffer pour récupérer les données :

Puis vous pouvez lire le flux en temps réel avec `int read(short[] audioData, int offsetInShorts, int sizeInShorts)` :

```

```

Enfin, il ne faut pas oublier de fermer le flux et de libérer la mémoire :

```

```

26.2.3. Prendre des photos

26.2.3.1. Demander à une autre application de le faire

La première chose que nous allons voir, c'est la solution de facilité : comment demander à une autre application de prendre des photos pour nous, puis ensuite les récupérer. On va bien entendu utiliser un intent, et son action sera `MediaStore.ACTION_IMAGE_CAPTURE`. Vous vous rappelez comment on lance une activité en lui demandant un résultat, j'espère ! Avec `void startActivityForResult(Intent intent, int requestCode)` où `requestCode` est un code qui permet d'identifier le retour. Le résultat sera ensuite disponible dans `void onActivityResult(int requestCode, int resultCode, Intent data)` avec `requestCode` qui vaut comme le `requestCode` que vous avez passé précédemment. On va ensuite préciser qu'on veut que l'image soit en extra dans le retour :

```

```

Il faut ensuite récupérer la photo dès que l'utilisateur revient dans l'application. On a ici un problème, parce que toutes les applications ne renverront pas le même résultat. Certaines renverront une image comme nous le voulons ; d'autres, juste une miniature… Nous allons donc voir ici comment gérer ces deux cas :

```

```

26.2.3.2. Tout gérer nous-mêmes

La technique précédente peut dépanner par moments, mais ce n'est pas non plus la solution à tout. Il se peut qu'on veuille avoir le contrôle total sur notre caméra ! Pour cela, on aura besoin de la permission de l'utilisateur d'utiliser sa caméra :

```

```

Vous pouvez ensuite manipuler très simplement la caméra avec la classe `Camera`. Pour récupérer une instance de cette classe, on utilise la méthode `static Camera Camera.open()`.

Il est ensuite possible de modifier les paramètres de l'appareil avec `void setParameters(Camera.Parameters params)`. Cependant, avant toute chose, il faut s'assurer que l'appareil peut supporter les paramètres qu'on va lui donner. En effet, chaque appareil aura un objectif photographique différent et par conséquent des caractéristiques différentes, alors il faudra faire en sorte de gérer le plus de cas possible. On va donc récupérer les paramètres avec `Camera.Parameters getParameters()`, puis on pourra vérifier les modes supportés par l'appareil avec différentes méthodes, par exemple :

```
```

Vous trouverez plus d'informations sur les modes supportés sur la page de `Camera.Parameters`. Une fois que vous connaissez les modes compatibles, vous pouvez manipuler la caméra à volonté :

```
```

Ensuite, il existe deux méthodes pour prendre une photo :

```
```

> À noter que la seconde méthode, celle avec `postview`, ne sera accessible que si vous avez activé la prévisualisation.

On rencontre ici deux types de classes appelées en *callback* :

- `Camera.ShutterCallback` est utilisée pour indiquer le moment exact où la photo est prise. Elle ne contient qu'une méthode, `void onShutter()`.
- `Camera.PictureCallback` est utilisée une fois que l'image est prête. Elle contient la méthode `void onPictureTaken(byte[] data, Camera camera)` avec l'image contenue dans `data` et la `camera` avec laquelle la photo a été prise.

Ainsi, `shutter` est lancé dès que l'image est prise, mais avant qu'elle soit prête. `raw` correspond à l'instant où l'image est prête mais pas encore traitée pour correspondre aux paramètres que vous avez entrés. Encore après sera appelé `postview`, quand l'image sera redimensionnée comme vous l'avez demandé (ce n'est pas supporté par tous les appareils). Enfin, `jpeg` sera appelé dès que l'image finale sera prête. Vous pouvez passer `null` à tous les *callbacks* si vous n'en avez rien à faire :

```
```

Enfin, on va voir comment permettre à l'utilisateur de prévisualiser ce qu'il va prendre en photo. Pour cela, on a besoin d'une vue particulière : `SurfaceView`. Il n'y a pas d'attributs particuliers à connaître pour la déclaration XML :

```

```

On aura ensuite besoin de récupérer le `SurfaceHolder` associé à notre `SurfaceView`, et ce avec la méthode `SurfaceHolder getHolder()`. On a ensuite besoin de lui attribuer un type, ce qui donne :

```

```

Ne vous inquiétez pas, c'est bientôt fini ! On n'a plus qu'à implémenter des méthodes de *callback* de manière à pouvoir gérer correctement le cycle de vie de la caméra et de la surface de prévisualisation. Pour cela, on utilise l'interface `SurfaceHolder.Callback` qui contient trois méthodes de *callback* qu'il est possible d'implémenter :

— `void surfaceChanged(SurfaceHolder holder, int format, int width, int height)` est lancée quand le `SurfaceView` change de dimensions.
— `void surfaceCreated(SurfaceHolder holder)` est appelée dès que la surface est créée. C'est dedans qu'on va associer la caméra au `SurfaceView`.
— À l'opposé, au moment de la destruction de la surface, la méthode `void surfaceDestroyed(SurfaceHolder holder)` sera exécutée. Elle permettra de dissocier la caméra et la surface.

Voici maintenant un exemple d'implémentation de cette synergie :

```

```

Enfin, pour libérer la caméra, on utilise la méthode `void release()`.

26.2.4. Enregistrer des vidéos

26.2.4.1. Demander à une autre application de le faire à notre place

Encore une fois, il est tout à fait possible de demander à une autre application de prendre une vidéo pour nous, puis de la récupérer afin de la traiter. Cette fois, l'action à spécifier est `MediaStore.ACTION_VIDEO_CAPTURE`. Pour préciser dans quel emplacement stocker la vidéo, il faut utiliser l'extra `MediaStore.EXTRA_OUTPUT` :

```

```

26.2.4.2. Tout faire nous-mêmes

Tout d'abord, on a besoin de trois autorisations : une pour utiliser la caméra, une pour enregistrer le son et une pour enregistrer la vidéo :

Au final, maintenant qu'on sait enregistrer du son, enregistrer de la vidéo n'est pas beaucoup plus complexe. En effet, on va encore utiliser `MediaRecorder`. Cependant, avant cela, il faut débloquer la caméra pour qu'elle puisse être utilisée avec le `MediaRecorder`. Il suffit pour cela d'appeler sur votre caméra la méthode `void unlock()`. Vous pouvez maintenant associer votre `MediaRecorder` et votre `Camera` avec la méthode `void setCamera(Camera camera)`. Puis, comme pour l'enregistrement audio, il faut définir les sources :

Cependant, quand on enregistre une vidéo, il est préférable de montrer à l'utilisateur ce qu'il est en train de filmer de manière à ce qu'il ne filme pas à l'aveugle. Comme nous l'avons déjà fait pour la prise de photographies, il est possible de donner un `SurfaceView` au `MediaRecorder`. La méthode à utiliser pour cela est `void setPreviewDisplay(SurfaceView surface)`. Encore une fois, vous pouvez implémenter les méthodes de *callback* contenues dans `SurfaceHolder.Callback`.

Enfin, comme pour l'enregistrement audio, on doit définir l'emplacement où enregistrer le fichier, préparer le lecteur, puis lancer l'enregistrement.

Toujours appeler `setPreviewDisplay` *avant* `prepare`, sinon vous aurez une erreur.

Enfin, il faut libérer la mémoire une fois la lecture terminée :

— Android est capable de lire nativement beaucoup de formats de fichier différents, ce qui en fait un lecteur multimédia mobile idéal.
— Pour lire des fichiers multimédia, on peut utiliser un objet `MediaPlayer`. Il s'agit d'un objet qui se comporte comme une machine à états, il est donc assez délicat et lourd à manipuler ; cependant, il permet de lire des fichiers efficacement dès qu'on a appris à le maîtriser.
— Pour afficher des vidéos, on devra passer par une `VideoView`, qu'il est possible de lier à un `MediaPlayer` auquel on donnera des fichiers vidéo qu'il pourra lire nativement.
— L'enregistrement sonore est plus délicat, il faut réfléchir à l'avance à ce qu'on va faire en fonction de ce qu'on désire faire. Par exemple, `MediaRecorder` est en général utilisé, mais si on veut quelque chose de moins lourd, sur lequel on peut effectuer des traitements en temps réel, on utilisera plutôt `AudioRecord`.

— Il est possible de prendre une photo avec `Camera`. Il est possible de personnaliser à l'extrême son utilisation pour celui qui désire contrôler tous les aspects de la prise d'images.
— Pour prendre des vidéos, on utilisera aussi un `MediaRecorder`, mais on fera en sorte d'afficher une prévisualisation du résultat grâce à un `SurfaceView`.

27. Les capteurs

La majorité des appareils modernes sont bien plus que de simples outils pour communiquer ou naviguer sur internet. Ils ont des capacités sensorielles, matérialisées par leurs capteurs. Ces capteurs nous fournissent des informations brutes avec une grande précision, qu'il est possible d'interpréter pour comprendre les transitions d'état que vit le terminal. On trouve par exemple des accéléromètres, des gyroscopes, des capteurs de champ magnétique, etc. Tous ces capteurs nous permettent d'explorer de nouvelles voies, d'offrir de nouvelles possibilités aux utilisateurs.

On va donc voir dans ce chapitre comment surveiller ces capteurs et comment les manipuler. On verra ainsi les informations que donnent les capteurs et comment en déduire ce que fait faire l'utilisateur à l'appareil.

27.1. Les différents capteurs

On peut répartir les capteurs en trois catégories :

— Les capteurs de mouvements : en mesurant les forces d'accélération et de rotation sur les trois axes, ces capteurs sont capables de déterminer dans quelle direction se dirige l'appareil. On y trouve l'accéléromètre, les capteurs de gravité, les gyroscopes et les capteurs de vecteurs de rotation.
— Les capteurs de position : évidemment, ils déterminent la position de l'appareil. On trouve ainsi les capteurs d'orientation et le magnétomètre.
— Les capteurs environnementaux : ce sont trois capteurs (baromètre, photomètre et thermomètre) qui mesurent la pression atmosphérique, l'illumination et la température ambiante.

D'un point de vue technique, on trouve deux types de capteurs. Certains sont des composants matériels, c'est-à-dire qu'il y a un composant physique présent sur le terminal. Ils fournissent des données en prenant des mesures. Certains autres capteurs sont uniquement présents d'une manière logicielle. Ils se basent sur des données fournies par des capteurs physiques pour calculer des données nouvelles.

Il n'est pas rare qu'un terminal n'ait pas tous les capteurs, mais seulement une sélection. Par exemple, la grande majorité des appareils ont un accéléromètre ou un magnétomètre, mais peu ont un thermomètre. De plus, il arrive qu'un terminal ait plusieurs exemplaires d'un capteur, mais calibrés d'une manière différente de façon à avoir des résultats différents.

Ces différents capteurs sont représentés par une valeur dans la classe `Sensor`. On trouve ainsi :

Nom du capteur	Valeur système	Type	Description	Utilisation typique

Accéléromètre	**TYPE_ACCELEROMETER**	**Matériel**	**Mesure la force d'accélération appliquée au terminal sur les trois axes (x, y et z), donc la force de gravitation (m/s^2).**	**Détecter les mouvements.**
Tous les capteurs	TYPE_ALL	Matériel et logiciel	Représente tous les capteurs qui existent.	
Gyroscope	*TYPE_GYROSCOPE*	*Matériel*	*Mesure le taux de rotation sur chacun des trois axes en radian par seconde (rad/s).*	*Détecter l'orientation de l'appareil.*
Photomètre	TYPE_LIGHT	Matériel	Mesure le niveau de lumière ambiante en lux (lx).	Détecter la luminosité pour adapter celle de l'écran de l'appareil.
Magnétomètre	TYPE_MAGNETIC_FIELD	Matériel	Mesure le champ géomagnétique sur les trois axes en microtesla (T).	Créer un compas.
Orientation	TYPE_ORIENTATION	Logiciel	Mesure le degré de rotation que l'appareil effectue sur les trois axes.	Déterminer la position de l'appareil.
Baromètre	*TYPE_PRESSURE*	*Matériel*	*Mesure la pression ambiante en hectopascal (hPa) ou millibar (mbar).*	*Surveiller les changements de pression de l'air ambiant.*
Capteur de proximité	TYPE_PROXIMITY	Matériel	Mesure la proximité d'un objet en centimètres (cm).	Détecter si l'utilisateur porte le téléphone à son oreille pendant un appel.
Thermomètre	TYPE_TEMPERATURE	Matériel	Mesure la température de l'appareil en degrés Celsius (°C).	Surveiller la température.

Les lignes en italique correspondent aux valeurs qui existent dans l'API 7 mais qui ne sont pas utilisables avant l'API 9.

27.2. Opérations génériques

27.2.1. Demander la présence d'un capteur

Il se peut que votre application n'ait aucun sens sans un certain capteur. Si c'est un jeu qui exploite la détection de mouvements par exemple, vous feriez mieux d'interdire aux gens qui n'ont pas un accéléromètre de pouvoir télécharger votre application sur le Play Store. Pour indiquer qu'on ne veut pas qu'un utilisateur sans accéléromètre puisse télécharger votre application, il vous faudra ajouter une ligne de type `<uses-feature>` dans votre Manifest :

```
```

N'oubliez pas que `android:required="true"` sert à préciser que la présence de l'accéléromètre est absolument indispensable. S'il est possible d'utiliser votre application sans l'accéléromètre mais qu'il est fortement recommandé d'en posséder un, alors il vous suffit de mettre à la place `android:required="false"`.

27.2.2. Identifier les capteurs

La classe qui permet d'accéder aux capteurs est `SensorManager`. Pour en obtenir une instance, il suffit de faire :

```
```

Comme je l'ai déjà dit, les capteurs sont représentés par la classe `Sensor`. Si vous voulez connaître la liste de tous les capteurs existants sur l'appareil, il vous faudra utiliser la méthode `List<Sensor> getSensorList(int type)` avec `type` qui vaut `Sensor.TYPE_ALL`. De même, pour connaître tous les capteurs qui correspondent à une catégorie de capteurs, utilisez l'une des valeurs vues précédemment dans cette même méthode. Par exemple, pour connaître la liste de tous les magnétomètres :

```
```

Il est aussi possible d'obtenir une instance d'un capteur. Il suffit d'utiliser la méthode `Sensor getDefaultSensor(int type)` avec `type` un identifiant présenté dans le tableau précédent. Comme je vous l'ai déjà dit, il peut y avoir plusieurs capteurs qui ont le même objectif dans un appareil, c'est pourquoi cette méthode ne donnera que l'appareil par défaut, celui qui correspondra aux besoins les plus génériques.

Si le capteur demandé n'existe pas dans l'appareil, la méthode `getDefaultSensor` renverra `null`.

```
```

Il est ensuite possible de récupérer des informations sur le capteur, comme par exemple sa consommation électrique avec `float getPower()` et sa portée avec `float getMaximumRange()`.

Vérifiez toujours qu'un capteur existe, même s'il est très populaire. Par exemple, il est peu probable qu'un accéléromètre soit absent, mais c'est possible !

27.2.3. Détection des changements des capteurs

L'interface `SensorEventListener` permet de détecter deux types de changement dans les capteurs :

- Un changement de précision du capteur avec la méthode de *callback* `void onAccuracyChanged(Sensor sensor, int accuracy)` avec `sensor` le capteur dont la précision a changé et `accuracy` la nouvelle précision. `accuracy` peut valoir `SensorManager.SENSOR_STATUS_ACCURACY_LOW` pour une faible précision, `SensorManager.SENSOR_STATUS_ACCURACY_MEDIUM` pour une précision moyenne, `SensorManager.SENSOR_STATUS_ACCURACY_HIGH` pour une précision maximale et `SensorManager.SENSOR_STATUS_ACCURACY_UNRELIABLE` s'il ne faut pas faire confiance à ce capteur.
- Le capteur a calculé une nouvelle valeur, auquel cas se lancera la méthode de *callback* `void onSensorChanged(SensorEvent event)`. Un `SensorEvent` indique à chaque fois quatre informations contenues dans quatre attributs : l'attribut `accuracy` indique la précision de cette mesure (il peut avoir les mêmes valeurs que précédemment), l'attribut `sensor` contient une référence au capteur qui a fait la mesure, l'attribut `timestamp` est l'instant en nanosecondes où la valeur a été prise, et enfin les valeurs sont contenues dans l'attribut `values`.

`values` est un tableau d'entiers. Si dans le tableau précédent j'ai dit que les valeurs correspondaient aux trois axes, alors le tableau a trois valeurs : `values[0]` est la valeur sur l'axe x, `values[1]` la valeur sur l'axe y et `values[2]` la valeur sur l'axe z. Si le calcul ne se fait pas sur trois axes, alors il n'y aura que `values[0]` qui contiendra la valeur. Attention, cette méthode sera appelée très souvent, il est donc de votre devoir de ne pas effectuer d'opérations bloquantes à l'intérieur. Si vous effectuez des opérations longues à résoudre, alors il se peut que la méthode soit à nouveau lancée alors que l'ancienne exécution n'avait pas fini ses calculs, ce qui va encombrer le processeur au fur et à mesure.

Une fois notre interface écrite, il faut déclarer au capteur que nous sommes à son écoute. Pour cela, on va utiliser la méthode `boolean registerListener(SensorEventListener listener, Sensor sensor, int rate)` de `SensorManager`, avec le `listener`, le capteur dans `sensor` et la fréquence de mise à jour dans `rate`. Il est possible de donner à `rate` les valeurs suivantes, de la fréquence la moins élevée à la plus élevée :

— `SensorManager.SENSOR_DELAY_NORMAL` (0,2 seconde entre chaque prise) ;
— `SensorManager.SENSOR_DELAY_UI` (0,06 seconde entre chaque mise à jour, délai assez lent qui convient aux interfaces graphiques) ;
— `SensorManager.SENSOR_DELAY_GAME` (0,02 seconde entre chaque prise, convient aux jeux) ;
— `SensorManager.SENSOR_DELAY_FASTEST` (0 seconde entre les prises).

Le délai que vous indiquez n'est qu'une indication, il ne s'agit pas d'un délai très précis. Il se peut que la prise se fasse avant ou après le moment choisi. De manière générale, la meilleure pratique est d'avoir la valeur la plus lente possible, puisque c'est elle qui permet d'économiser le plus le processeur et donc la batterie.

Enfin, on peut désactiver l'écoute d'un capteur avec `void unregisterListener(SensorEventListener listener, Sensor sensor)`. N'oubliez pas de désactiver vos capteurs pendant que l'activité n'est pas au premier plan (donc il faut le désactiver pendant `onPause()` et le réactiver pendant `onResume()`), car le système ne le fera pas pour vous. De manière générale, désactivez les capteurs dès que vous ne les utilisez plus.

27.3. Les capteurs de mouvements

On va ici étudier les capteurs qui permettent de garder un œil sur les mouvements du terminal. Pour l'API 7, on trouve surtout l'accéléromètre, mais les versions suivantes supportent aussi le gyroscope, ainsi que trois capteurs logiciels (gravitationnel, d'accélération linéaire et de vecteurs de rotation). De nos jours, on trouve presque tout le temps un accéléromètre et un gyroscope. Pour les capteurs logiciels, c'est plus complexe puisqu'ils se basent souvent sur plusieurs capteurs pour déduire et calculer des données, ils nécessitent donc la présence de plusieurs capteurs différents.

On utilise les capteurs de mouvements pour détecter les… mouvements. Je pense en particulier aux inclinaisons, aux secousses, aux rotations ou aux balancements. Typiquement, les capteurs de mouvements ne sont pas utilisés pour détecter la position de l'utilisateur, mais si on les utilise conjointement avec d'autres capteurs, comme par exemple le magnétomètre, ils permettent de mieux évaluer ses déplacements.

Tous ces capteurs retournent un tableau de `float` de taille 3, chaque élément correspondant à un axe différent (voir figure suivante). La première valeur, `values[0]`, se trouve sur l'axe x, il s'agit de l'axe de l'horizon, quand vous bougez votre téléphone de gauche à droite. Ainsi, la

valeur est positive et augmente quand vous déplacez le téléphone vers la droite, alors qu'elle est négative et continue à diminuer plus vous le déplacez vers la gauche.

La deuxième valeur, `values[1]`, correspond à l'axe y, c'est-à-dire l'axe vertical, quand vous déplacez votre téléphone de haut en bas. La valeur est positive quand vous déplacez le téléphone vers le haut et négative quand vous le déplacez vers le bas.

Enfin, la troisième valeur, `values[2]`, correspond à l'axe z, il s'agit de l'axe sur lequel vous pouvez éloigner ou rapprocher le téléphone de vous. Quand vous le rapprochez de vous, la valeur est positive et, quand vous l'éloignez, la valeur est négative.

Vous l'aurez compris, toutes ces valeurs respectent un schéma identique : un 0 signifie pas de mouvement, une valeur positive un déplacement dans le sens de l'axe et une valeur négative un déplacement dans le sens inverse de celui de l'axe.

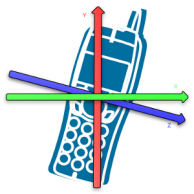

FIGURE 27.1. – Les différents axes

Enfin, cela va peut-être vous sembler logique, mais l'accéléromètre ne mesure pas du tout la vitesse, juste le changement de vitesse. Si vous voulez obtenir la vitesse depuis les données de l'accéléromètre, il vous faudra intégrer l'accélération sur le temps (que l'on peut obtenir avec l'attribut `timestamp`) pour obtenir la vitesse. Et pour obtenir une distance, il vous faudra intégrer la vitesse sur le temps.

27.4. Les capteurs de position

On trouve trois (bon, en fait deux et un autre moins puissant) capteurs qui permettent de déterminer la position du terminal : le magnétomètre, le capteur d'orientation et le capteur de proximité (c'est le moins puissant, il est uniquement utilisé pour détecter quand l'utilisateur a le visage collé au téléphone, afin d'afficher le menu uniquement quand l'utilisateur n'a pas le téléphone contre la joue). Le magnétomètre et le capteur de proximité sont matériels, alors que le capteur d'orientation est une combinaison logicielle de l'accéléromètre et du magnétomètre.

Le capteur d'orientation et le magnétomètre renvoient un tableau de taille 3, alors que pour le capteur de proximité c'est plus compliqué. Parfois il renvoie une valeur en centimètres, parfois juste une valeur qui veut dire « proche » et une autre qui veut dire « loin » ; dans ces cas-là, un objet est considéré comme éloigné s'il se trouve à plus de 5 cm.

Cependant, le magnétomètre n'est pas utilisé que pour déterminer la position de l'appareil. Si on l'utilise conjointement avec l'accéléromètre, il est possible de détecter l'inclinaison de l'appareil. Et pour cela, la seule chose dont nous avons besoin, c'est de faire de gros calculs trigonométriques. Enfin… Android va (heureusement !) les faire pour nous.

Dans tous les cas, nous allons utiliser deux capteurs, il nous faudra donc déclarer les deux dans deux listeners différents. Une fois les données récupérées, il est possible de calculer ce qu'on

appelle la méthode `Rotation` avec la méthode statique `static boolean SensorManager.getRotationMatrix(float[] R, float[] I, float[] gravity, float[] geomagnetic)` avec `R` le tableau de taille 9 dans lequel seront stockés les résultats, `I` un tableau d'inclinaison qui peut bien valoir `null`, `gravity` les données de l'accéléromètre et `geomagnetic` les données du magnétomètre.

La matrice rendue s'appelle une matrice de rotation. À partir de celle-ci, vous pouvez obtenir l'orientation de l'appareil avec `static float[] SensorManager.getOrientation(float[] R, float[] values)` avec `R` la matrice de rotation et `values` le tableau de taille 3 qui contiendra la valeur de la rotation pour chaque axe :

```
```

Je vais vous expliquer maintenant à quoi correspondent ces chiffres, cependant avant toute chose, vous devez imaginez votre téléphone portable posé sur une table, le haut qui pointe vers vous, l'écran vers le sol. Ainsi, l'axe z pointe de bas en haut, l'axe x de droite à gauche et l'axe y de "loin devant vous" à "loin derrière vous" (en gros c'est l'axe qui vous traverse) :

— La rotation sur l'axe z est le mouvement que vous faites pour ouvrir ou fermer une bouteille de jus d'orange posée verticalement sur une table. C'est donc comme si vous englobiez le téléphone dans votre main et que vous le faisiez tourner sur l'écran. Il s'agit de l'angle entre le nord magnétique et l'angle de l'axe y. Il vaut 0 quand l'axe y pointe vers le nord magnétique, 180 si l'axe y pointe vers le sud, 90 quand il pointe vers l'est et 270 quand il pointe vers l'ouest.
— La rotation autour de l'axe x est le mouvement que vous faites quand vous ouvrez la bouteille de jus d'orange posée sur une table mais que le bouchon pointe vers votre droite. Enfin, ce n'est pas malin parce que vous allez tout renverser par terre. Elle vaut -90 quand vous tournez vers vous (ouvrir la bouteille) et 90 quand vous tournez dans l'autre sens (fermer la bouteille).
— La rotation sur l'axe y est le mouvement que vous faites quand vous ouvrez une bouteille de jus d'orange couchée sur la table alors que le bouchon pointe vers vous. Elle vaut 180 quand vous tournez vers la gauche (fermer la bouteille) et -180 quand vous tournez vers la droite (ouvrir la bouteille).

Enfin, il vous est possible de changer le système de coordonnées pour qu'il corresponde à vos besoins. C'est utile si votre application est censée être utilisée en mode paysage plutôt qu'en mode portrait par exemple. On va utiliser la méthode `static boolean SensorManager.remapCoordinateSystem(float[] inR, int X, int Y, float[] outR)` avec `inR` la matrice de rotation à transformer (celle qu'on obtient avec `getRotationMatrix`), `X` désigne la nouvelle orientation de l'axe x, `Y` la nouvelle orientation de l'axe y et `outR` la nouvelle matrice de rotation (ne mettez pas `inR` dedans). Vous pouvez mettre dans `X` et `Y` des valeurs telles que `SensorManager.AXIS_X` qui représente l'axe x et `SensorManager.AXIS_MINUS_X` son orientation inverse. Vous trouverez de même les valeurs `SensorManager.AXIS_Y`, `SensorManager.AXIS_MINUS_Y`, `SensorManager.AXIS_Z` et `SensorManager.AXIS_MINUS_Z`.

27.5. Les capteurs environnementaux

Pour être franc, il n'y a pas tellement à dire sur les capteurs environnementaux. On en trouve trois dans l'API 7 : le baromètre, le photomètre et le thermomètre. Ce sont tous des capteurs matériels, mais il est bien possible qu'ils ne soient pas présents dans un appareil. Tout dépend du type d'appareil, pour être exact. Sur un téléphone et sur une tablette, on en trouve rarement (à l'exception du photomètre qui permet de détecter automatiquement la meilleure luminosité pour l'écran), mais sur une station météo ils sont souvent présents. Il faut donc redoubler de prudence quand vous essayez de les utiliser, vérifiez à l'avance leur présence.

Tous ces capteurs rendent des valeurs uniques, pas de tableaux à plusieurs dimensions.

— La majorité des appareils sous Android utilisent des capteurs pour créer un lien supplémentaire entre l'utilisateur et son appareil. On rencontre trois types de capteurs : les capteurs de mouvements, les capteurs de position et les capteurs environnementaux.
— La présence d'un capteur dépend énormément des appareils, il faut donc faire attention à bien déclarer son utilisation dans le Manifest et à vérifier sa présence au moment de l'utilisation.
— Les capteurs de mouvements permettent de détecter les déplacements que l'utilisateur fait faire à son appareil. Il arrive qu'ils soient influencés par des facteurs extérieurs comme la gravité, il faut donc réfléchir à des solutions basées sur des calculs physiques quand on désire avoir une valeur précise d'une mesure.
— Les capteurs de position sont capables de repérer la position de l'appareil par rapport à un référentiel. Par exemple, le capteur de proximité peut donner la distance entre l'utilisateur et l'appareil. En pratique, le magnétomètre est surtout utilisé conjointement avec des capteurs de mouvements pour détecter d'autres types de mouvements, comme les rotations.
— Les capteurs environnementaux sont vraiment beaucoup plus rares sur un téléphone ou une tablette, mais il existe d'autres terminaux spécifiques, comme des stations météorologiques, qui sont bardés de ce type de capteurs.

28. TP : un labyrinthe

Nous voici arrivés au dernier TP de ce cours ! Et comme beaucoup de personnes m'ont demandé comment faire un jeu, je vais vous indiquer ici quelques pistes de réflexion en créant un jeu relativement simple : un labyrinthe. Et en dépit de l'apparente simplicité de ce jeu, vous verrez qu'il faut penser à beaucoup de choses pour que le jeu reste amusant et cohérent.

Nous nous baserons ici uniquement sur les API que nous connaissons déjà. Ainsi, ce TP n'aborde pas Open GL par exemple, dont la maîtrise va bien au-delà de l'objectif de ce cours ! Mais vous verrez qu'avec un brin d'astuce il est déjà possible de faire beaucoup avec ce que nous avons à portée de main.

28.1. Objectifs

Vous l'aurez compris, nous allons faire un labyrinthe. Le principe du jeu est très simple : le joueur utilise l'accéléromètre de son téléphone pour diriger une boule. Ainsi, quand il penche l'appareil vers le bas, la boule se déplace vers le bas. Quand il penche l'appareil vers le haut, la boule se dirige vers le haut, de même pour la gauche et la droite. L'objectif est de pouvoir placer la boule à un emplacement particulier qui symbolisera la sortie. Cependant, le parcours sera semé d'embûches ! Il faudra en effet faire en sorte de zigzaguer entre des trous situés dans le sol, placés par les immondes Zörglubienotchs qui n'ont qu'un seul objectif : détruire le monde (Ha ! Ha ! Ha ! Ha !).

Le scénario est optionnel.

La figure suivante est un aperçu du résultat final que j'obtiens.

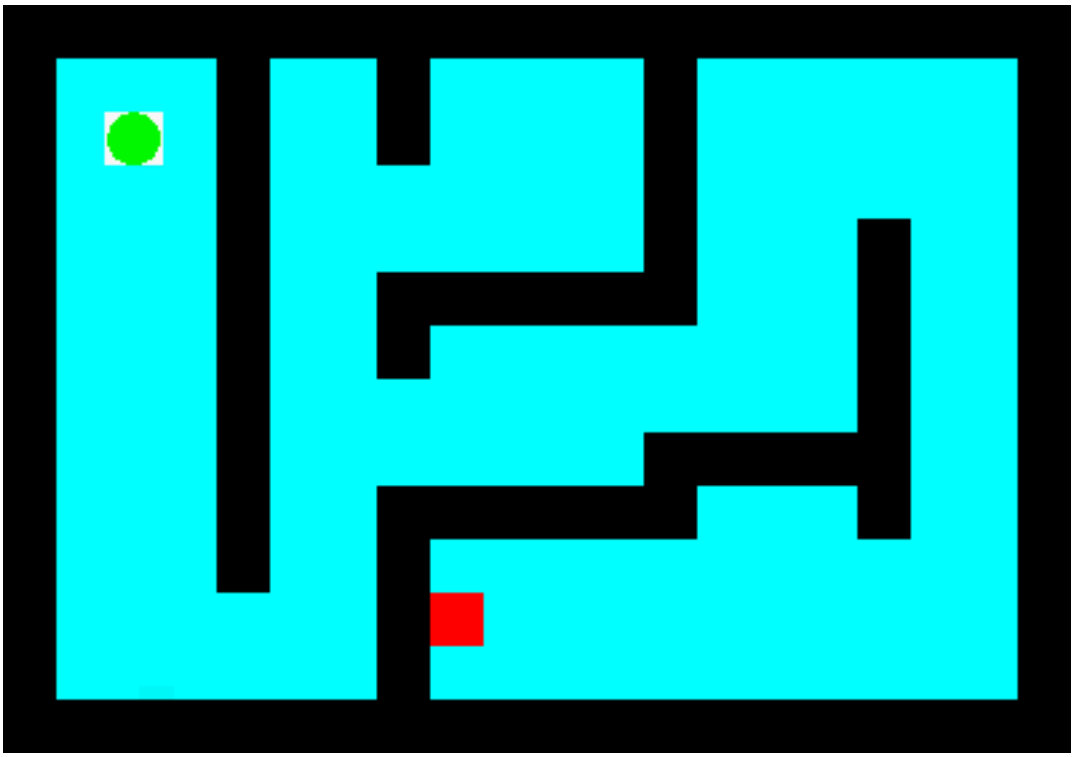

FIGURE 28.1. – Le labyrinthe

On peut y voir les différents éléments qui composent le jeu :

— La boule verte, le seul élément qui bouge quand vous bougez votre téléphone.
— Une case blanche, qui indique le départ du labyrinthe.
— Une case rouge, qui indique l'objectif à atteindre pour détruire le roi des Zörglubienotchs.
— Plein de cases noires : ce sont les pièges posés par les Zörglubienotchs et qui détruisent votre boule.

Quand l'utilisateur perd, une boîte de dialogue le signale et le jeu se met en pause. Quand l'utilisateur gagne, une autre boîte de dialogue le signale et le jeu se met en pause, c'est aussi simple que cela !

Avant de vous laisser vous aventurer seuls, laissez-moi vous donner quelques indications qui pourraient vous être précieuses.

28.2. Spécifications techniques

28.2.1. Organisation du code

De manière générale, quand on développe un jeu, on doit penser à trois moteurs qui permettront de gérer les différentes composantes qui constituent le jeu :

— Le moteur graphique qui s'occupera de dessiner.
— Le moteur physique qui s'occupera de gérer les positions, déplacements et interactions entre les éléments.
— Le moteur multimédia qui joue les animations et les sons au bon moment.

Nous n'utiliserons que deux de ces moteurs : le moteur graphique et le moteur physique. Cette organisation implique une chose : il y aura deux représentations pour chaque élément. Par exemple, une représentation graphique de la boule — celle que connaîtra le moteur graphique — et une représentation physique — celle que connaîtra le moteur physique. On peut ainsi dire que la boule sera divisée en deux parties distinctes, qu'il faudra lier pour avoir un ensemble cohérent.

La toute première chose à laquelle il faut penser, c'est qu'on va donner du matériel à ces moteurs. Le moteur graphique ne peut dessiner s'il n'a rien à dessiner, le moteur physique ne peut calculer de déplacements s'il n'y a pas quelque chose qui bouge ! On va ainsi définir des modèles qui vont contenir les différentes informations sur les constituants.

28.2.2. Les modèles

Comme je viens de le dire, un modèle sera une classe Java qui contiendra des informations sur les constituants du jeu. Ces informations dépendront bien entendu de l'objet représenté. Réfléchissons maintenant à ce qui constitue notre jeu. Nous avons déjà une boule. Ensuite, nous avons des trous dans lesquels peut tomber la boule, une case de départ et une case d'arrivée. Ces trois types d'objets ne bougent pas, et se dessinent toujours un peu de la même manière ! On peut alors décréter qu'ils sont assez similaires quand même. Voyons maintenant ce que doivent contenir les modèles.

28.2.2.1. La boule

Il s'agit du cœur du jeu, de l'élément le plus compliqué à gérer. Tout d'abord, il va se déplacer, il nous faut donc connaître sa position. Le `Canvas` du `SurfaceView` se comporte comme n'importe quel autre `Canvas` que nous avons vu, c'est-à-dire qu'il possède un axe x qui va de gauche à droite (le rebord gauche vaut 0 et le rebord droit vaut la taille de l'écran en largeur). Il possède aussi un axe y qui va de haut en bas (le plafond du téléphone vaut 0 et le plancher vaut la taille de l'écran en hauteur). Vous aurez donc besoin de deux attributs pour situer votre boule sur le `Canvas` : un pour l'axe x, un pour l'axe y.

En plus de la position, il faut penser à la vitesse. Eh oui, plus la boule roule, plus elle accélère ! Comme notre boule se déplace sur deux axes (x et y), on aura besoin de deux indicateurs de vitesse : un pour l'axe x, et un pour l'axe y. Alors, accélérer, c'est bien, mais si notre boule dépasse la vitesse du son, c'est moins pratique pour jouer quand même. Il nous faudra alors aussi un attribut qui indiquera la vitesse à ne pas dépasser.

Pour le dessin, nous aurons aussi besoin d'indiquer la taille de la boule ainsi que sa couleur. De cette manière, on a pensé à tout, on obtient alors cette classe :

28.2.2.2. Les blocs

Même s'ils ont un comportement physique similaire, les blocs ont tous un dessin et un objectif différent. Il nous faut ainsi un moyen de les différencier, en dépit du fait qu'ils soient tous des objets de la classe `Bloc`. Alors comment faire ? Il existe deux solutions :

— Soit on crée des classes qui dérivent de `Bloc` pour chaque type de bloc, auquel cas on pourra tester si un objet appartient à une classe particulière avec l'instruction `instanceof`. Par exemple, `bloc instanceof sdz.chapitreCinq.labyrinthe.Trou`.
— Ou alors on ajoute un attribut `type` à la classe `Bloc`, qui contiendra le type de notre bloc. Tous les types possibles seront alors décrits dans une énumération.

J'ai privilégié la seconde méthode, tout simplement parce qu'elle impliquait d'utiliser les énumérations, ce qui en fait un exemple pédagogiquement plus intéressant.

? C'est quoi une énumération ?

Avec la programmation orientée objet, on utilise plus rarement les énumérations, et pourtant elles sont pratiques ! Une énumération, c'est une façon de décrire une liste de constantes. Il existe trois types de blocs (trou, départ, arrivée), on aura donc trois types de constantes dans notre énumération :

```
```

Comme vous pouvez le voir, on n'a pas besoin d'ajouter une valeur à nos constantes ; en effet, leur nom fera office de valeur.

Autre chose : comme il faut placer les blocs, nous avons encore une fois besoin des coordonnées du bloc. De plus, il est nécessaire de définir la taille d'un bloc. De ce fait, on obtient :

```
```

Comme vous pouvez le voir, j'ai fait en sorte qu'un bloc ait deux fois la taille de la boule.

28.2.3. Le moteur graphique

Très simple à comprendre, il sera en charge de dessiner les composants de notre scène de jeu. Ce à quoi il faut faire attention ici, c'est que certains éléments se déplacent (je pense en particulier à la boule). Il faut ainsi faire en sorte que le dessin corresponde toujours à la position exacte de l'élément : il ne faut pas que la boule se trouve à un emplacement et que le dessin affiche toujours son ancien emplacement. Regardez la figure suivante.

FIGURE 28.2. – À gauche le dessin de la boule, à droite sa représentation physique

Maintenant, regardez la figure suivante.

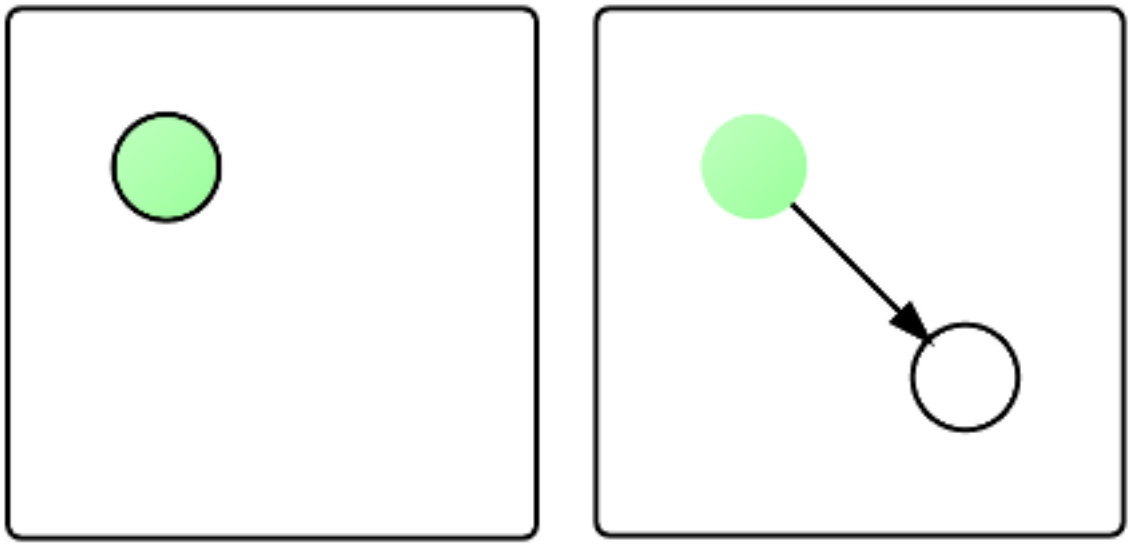

FIGURE 28.3. – Représentation des deux moteurs au temps T à gauche, T+1 à droite

À gauche, les deux représentations se superposent : la boule ne bouge pas, alors, au moment de dessiner la boule, il suffit de la dessiner au même endroit que précédemment. Cependant, à l'instant suivant (à droite), le joueur penche l'appareil, et la boule se met à se déplacer. On peut voir que la représentation graphique est restée au même endroit alors que la représentation physique a bougé, et donc ce que le joueur voit n'est pas ce que le jeu sait de l'emplacement de la boule. C'est ce que je veux dire par « il faut faire en sorte que le dessin corresponde toujours à la position exacte de l'élément ». Ainsi, à chaque fois que vous voulez dessiner la boule, il faudra le faire avec sa position exacte.

Pour effectuer les dessins, on va utiliser un `SurfaceView`, puisqu'il s'agit de la manière la plus facile de dessiner avec de bonnes performances. Ensuite, chaque élément devra être dessiné sur le `Canvas` du `SurfaceView`. Par exemple, chez moi, la boule est un disque de rayon 10 et de couleur verte.

Pour vous faciliter la vie, je vous propose de récupérer tout simplement le *framework* que nous avons écrit dans le chapitre sur le dessin, puisqu'il convient parfaitement à ce projet. Il ne vous reste plus ensuite qu'à dessiner dans la méthode de *callback* `void onDraw(Canvas canvas)`.

Pour adapter le dessin à tous les périphériques, vos éléments doivent être proportionnels à la taille de l'écran. Je pense au moins aux différents blocs qui doivent rentrer dans tous les écrans, même les plus petits.

28.2.4. Le moteur physique

Plus délicat à gérer que le moteur graphique, le moteur physique gère la position, les déplacements et l'interaction entre les différents éléments de votre jeu. De plus, dans notre cas particulier, il faudra aussi manipuler l'accéléromètre ! Vous savez déjà le faire normalement, alors pas de soucis ! Cependant, qu'allons-nous faire des données fournies par le capteur ? Eh bien, nous n'avons besoin que de deux données : les deux axes. J'ai choisi de faire en sorte que la position de base soit le téléphone posé à plat sur une table. Quand l'utilisateur penche le téléphone vers lui, la boule « tombe », comme si elle était attirée par la gravité. Si l'utilisateur penche l'appareil dans l'autre sens quand la boule « tombe », alors elle remonte une pente, elle a du mal à « monter » et elle se met à rouler dans le sens de la pente, comme le ferait une vraie boule. De ce fait, j'ai conservé les données sur deux axes seulement : x et y.

Ces données servent à modifier la vitesse de la boule. Si la boule roule dans le sens de la pente, elle prend de la vitesse et donc sa vitesse augmente avec la valeur du capteur. Si la vitesse dépasse la vitesse maximale, alors on impose la vitesse maximale comme vitesse de la boule. Enfin, si la vitesse est négative… cela veut tout simplement dire que la boule se dirige vers la gauche ou le haut, c'est normal !

```

```

Maintenant que notre boule bouge, que faire quand elle rencontre un bloc ? Comment détecter cette rencontre ? Le plus simple est encore d'utiliser des objets de type `RectF`, tout simplement parce qu'ils possèdent une méthode qui permet de détecter si deux `RectF` entrent en collision. Cette méthode est `boolean intersect(RectF r)` : le `boolean` retourné vaudra `true` si les deux rectangles entrent bien en collision et `r` sera remplacé par le rectangle formé par la collision.

Je le répète, le rectangle passé en attribut sera modifié par cette méthode, il vous faut donc faire une copie du rectangle dont vous souhaitez vérifier la collision, sinon il sera modifié. Pour copier un `RectF`, utilisez le constructeur `public RectF(RectF r)`.

Ainsi, on va rajouter un rectangle à nos blocs et à notre boule. C'est très simple, il vous suffit de deux données : les coordonnées du point en haut à gauche (sur l'axe x et l'axe y), puis la taille du rectangle. Avec ces données, on peut très bien construire un rectangle, voyez vous-mêmes :

```

```

En fait, l'attribut `left` correspond à la coordonnée sur l'axe x du côté gauche du rectangle, `top` à la coordonnée sur l'axe y du plafond, `right` à la coordonnée sur l'axe y du côté droit et `bottom` à la coordonnée sur l'axe y du plancher. De ce fait, avec les données que je vous ai demandées, il suffit de faire :

```

```

Mais comment faire pour la boule ? C'est un disque, pas un rectangle !

Cela peut sembler bizarre, mais on n'a nullement besoin d'une représentation exacte de la boule, on peut accompagner sa représentation d'un rectangle, tout simplement parce que la majorité des collisions ne peuvent pas se faire en diagonale, uniquement sur les rebords extrêmes de la boule, comme schématisé à la figure suivante.

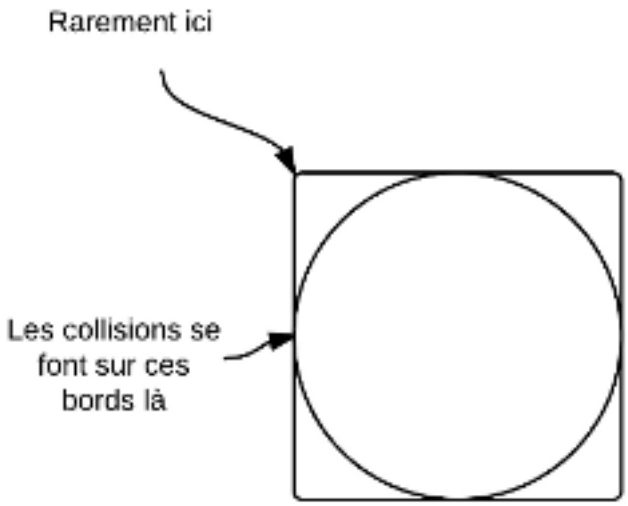

FIGURE 28.4. – Emplacement des collisions

Bien sûr, les collisions qui se feront sur les diagonales ne seront pas précises, mais franchement elles sont tellement rares et ce serait tellement complexe de les gérer qu'on va simplement les laisser tomber. De ce fait, il faut ajouter un `RectF` dans les attributs de la boule et, à chaque fois qu'elle bouge, il faut mettre à jour les coordonnées du rectangle pour qu'il englobe bien la boule et puisse ainsi détecter les collisions.

28.2.5. Le labyrinthe

C'est très simple, pour cette version simplifiée, le labyrinthe sera tout simplement une liste de blocs qui est générée au lancement de l'application. Chez moi, j'ai utilisé le labyrinthe suivant :

```
```

Comme vous pouvez le voir, ma méthode pour construire un bloc est simple, j'ai besoin de :

- Son type (`TROU`, `DEPART` ou `ARRIVEE`).
- Sa position sur l'axe x (attention, sa position en blocs et pas en pixels. Par exemple, si je mets 5, je parle du cinquième bloc, pas du cinquième pixel).
- Sa position sur l'axe y (en blocs aussi).

28.3. Ma solution

28.3.1. Le Manifest

La première chose à faire est de modifier le Manifest. Vous verrez deux choses particulières :

— L'appareil est bloqué en mode paysage (`<activity android:configChanges="orientation" android:screenOrientation="landscape" >`).
— L'application n'est pas montrée aux utilisateurs qui n'ont pas d'accéléromètre (`<uses-feature android:name="android.hardware.sensor.accelerometer" android:required="true" />`).

```

```

28.3.2. Les modèles

Nous allons tout d'abord voir les différents modèles qui permettent de décrire les composants de notre jeu.

28.3.2.1. Les blocs

```

```

Rien de spécial ici, je vous ai déjà parlé de tout auparavant. Remarquez le calcul qui permet de placer un bloc en fonction de sa position en tant que bloc et non en pixels.

28.3.2.2. La boule

```

```

28.3.3. Le moteur graphique

```

```

Rien de formidable ici non plus, on se contente de reprendre le *framework* et d'ajouter les dessins dedans.

28.3.4. Le moteur physique

28.3.5. L'activité

Télécharger le projet

28.4. Améliorations envisageables

28.4.1. Proposer plusieurs labyrinthes

Ce projet est quand même très limité, il ne propose qu'un labyrinthe. Avouons que jouer au même labyrinthe *ad vitam aeternam* est assez ennuyeux. On va alors envisager un système pour charger plusieurs labyrinthes. La première chose à faire, c'est de rajouter un modèle pour les labyrinthes. Il contiendra au moins une liste de blocs, comme précédemment :

Il suffira ensuite de passer le labyrinthe aux moteurs et de tout réinitialiser. Ainsi, on redessinera le labyrinthe, on cherchera le nouveau départ et on y placera la boule.

Enfin, si on fait cela, notre problème n'est pas vraiment résolu. C'est vrai qu'on pourra avoir plusieurs labyrinthes et qu'on pourra alterner entre eux, mais si on doit créer chaque fois un labyrinthe bloc par bloc, cela risque d'être quand même assez laborieux. Alors, comment créer un labyrinthe autrement ?

Une solution élégante serait d'avoir les labyrinthes enregistrés sur un fichier de façon à n'avoir qu'à le lire pour récupérer un labyrinthe et le partager avec le monde. Imaginons un peu comment fonctionnerait ce système. On pourrait avoir un fichier texte et chaque caractère correspondrait à un type de bloc. Par exemple :

— `o` serait un trou ;
— `d`, le départ ;
— `a`, l'arrivée.

Si on envisage ce système, le labyrinthe précédent donnerait ceci :

C'est tout de suite plus graphique, plus facile à développer, à entretenir et à déboguer. Pour transformer ce fichier texte en labyrinthe, il suffit de créer une boucle qui lira le fichier caractère par caractère, puis qui créera un bloc en fonction de la présence ou non d'un caractère à l'emplacement lu :

```
```

Pour les plus motivés d'entre vous, il est possible aussi de développer un éditeur de niveaux. Imaginez, vous possédez un menu qui permet de choisir le bloc à ajouter, puis il suffira à l'utilisateur de cliquer à l'endroit où il voudra que le bloc se place.

Vérifiez toujours qu'un labyrinthe a un départ et une arrivée, sinon l'utilisateur va tourner en rond pendant des heures ou n'aura même pas de boule !

28.4.2. Ajouter des sons

Parce qu'un peu de musique et des effets sonores permettent d'améliorer l'immersion. Enfin, si tant est qu'on puisse avoir de l'immersion dans ce genre de jeux avec de si jolis graphismes… Bref, il existe deux types de sons que devrait jouer notre jeu :

— Une musique de fond ;
— Des effets sonores. Par exemple, quand la boule de l'utilisateur tombe dans un trou, cela pourrait être amusant d'avoir le son d'une foule qui le hue.

Pour la musique, c'est simple, vous savez déjà le faire ! Utilisez un `MediaPlayer` pour jouer la musique en fond, ce n'est pas plus compliqué que cela. Si vous avez plusieurs musiques, vous pouvez aussi très bien créer une liste de lecture et passer d'une chanson à l'autre dès que la lecture d'une piste est terminée.

Pour les effets sonores, c'est beaucoup plus subtil. On va plutôt utiliser un `SoundPool`. En effet, il est possible qu'on ait à jouer plusieurs effets sonores en même temps, ce que `MediaPlayer` ne gère pas correctement ! De plus, `MediaPlayer` est lourd à utiliser, et on voudra qu'un effet sonore soit plutôt réactif. C'est pourquoi on va se pencher sur `SoundPool`.

Contrairement à `MediaPlayer`, `SoundPool` va devoir précharger les sons qu'il va jouer au lancement de l'application. Les sons vont être convertis en un format que supportera mieux Android afin de diminuer la latence de leur lecture. Pour les plus minutieux, vous pouvez même gérer le nombre de flux audio que vous voulez en même temps. Si vous demandez à `SoundPool` de jouer un morceau de plus que vous ne l'avez autorisé, il va automatiquement fermer un flux précédent, généralement le plus ancien. Enfin, vous pouvez aussi préciser une priorité manuellement pour gérer les flux que vous souhaitez garder. Par exemple, si vous jouez la musique dans un `SoundPool`, il faudrait pouvoir la garder quoi qu'il arrive, même si le nombre de flux autorisés est dépassé. Vous pouvez donc donner à la musique de fond une grosse priorité pour qu'elle ne soit pas fermée.

Ainsi, le plus gros défaut de cette méthode est qu'elle prend du temps au chargement. Vous devez insérer chaque son que vous allez utiliser avec la méthode `int load(String path, int`

`priority)`, `path` étant l'emplacement du son et `priority` la priorité que vous souhaitez lui donner (0 étant la valeur la plus basse possible). L'entier retourné sera l'identifiant de ce son, gardez donc cette valeur précieusement.

Si vous avez plusieurs niveaux, et que chaque niveau utilise un ensemble de sons différents, il est important que le chargement des sons se fasse en parallèle du chargement du niveau (dans un thread, donc) et surtout tout au début, pour que le chargement ne soit pas trop retardé par ce processus lent.

Une fois le niveau chargé, vous pouvez lancer la lecture d'un son avec la méthode `int play(int soundID, float leftVolume, float rightVolume, int priority, int loop, float rate)`, les paramètres étant :

— En tout premier l'identifiant du son, qui vous a été donné par la méthode `load()`.
— Le volume à gauche et le volume à droite, utile pour la lecture en stéréo. La valeur la plus basse est 0, la plus haute est 1.
— La priorité de ce flux. 0 est le plus bas possible.
— Le nombre de fois que le son doit être répété. On met 0 pour jamais, -1 pour toujours, toute autre valeur positive pour un nombre précis.
— Et enfin la vitesse de lecture. 1.0 est la vitesse par défaut, 2.0 sera deux fois plus rapide et 0.5 deux fois plus lent.

La valeur retournée est l'identifiant du flux. C'est intéressant, car cela vous permet de manipuler votre flux. Par exemple, vous pouvez arrêter un flux avec `void pause(int streamID)` et le reprendre avec `void resume(int streamID)`.

Enfin, une fois que vous avez fini un niveau, il vous faut appeler la méthode `void release()` pour libérer la mémoire, en particulier les sons retenus en mémoire. La référence au `SoundPool` vaudra `null`. Il vous faut donc créer un nouveau `SoundPool` par niveau, cela vous permet de libérer la mémoire entre chaque chargement.

Créer le moteur graphique et physique du jeu requiert beaucoup de temps et d'effort. C'est pourquoi il est souvent conseillé de faire appel à des moteurs préexistants comme AndEngine par exemple, qui est gratuit et *open source*. Son utilisation sort du cadre de ce cours ; cependant, si vous voulez faire un jeu, je vous conseille de vous y pencher sérieusement.