Le B.A. BA du développement mobile Android

Préambule *: L'objectif de ce petit projet est de tester quelques fonctionnalités de base de la programmation Android. Le look n'est pas essentiel.*
Une fois les tests terminés, vous devez trouver une idée d'application personnelle qui met en oeuvre les différents concepts étudiés (entre autres), application qui sera testée et notée par vos enseignants.
Si vous êtes à court d'idée originale, vous pouvez faire une nouvelle application à partir de l'étape 12 en enrichissant ce qui est proposé ici.

Etape 0 : Un nouveau projet

Dans IntelliJ, créer un nouveau projet Android en lui choisissant :

- un nom commençant par une majuscule : **PeleMele**
- un nom de domaine : celui proposé convient
- un nom de package : celui proposé convient
- un emplacement sur le disque
- Choix du SDK : **API23**
- Choix de l'activité : **EmptyActivity**
- Customize : ne rien changer

Attention de bien attendre que la création du projet soit terminée.

Etape 1 : Première exécution

Lorsque le projet est ouvert, lancer l'exécution sur la tablette (un peu long aussi la première fois). Confirmer le choix du support (détecté par défaut). L'activité principale affiche **Hello world!**, avec le nom du projet dans la barre du haut.
Pour lancer l'exécution sur un téléphone Android, il faut activer le mode développeur (quelque part dans les paramètres système du téléphone …)

Il est important de se familiariser avec les différents répertoires et fichiers.

- **java** : les fichiers sources Java
- **res** : fichiers XML des ressources
  - **res/drawable** : les images
  - **res/layout** : les interfaces graphiques des activités
  - **res/mipmap** : les icônes
  - **res/values/strings** : les chaînes de caractères
- **build.gradle** : fichier de configuration
- **AndroidManifest.xml** : fichier de description de l'application

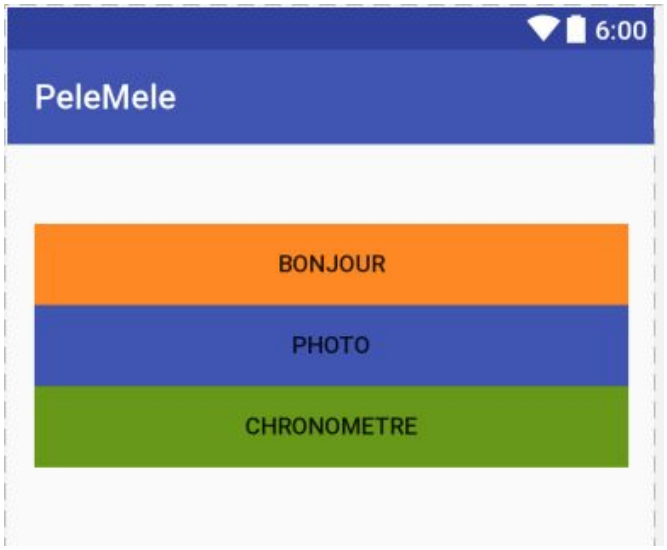
Etape 2 : Une (jolie) icône pour notre application

Choisir une image qui va servir d'icône à l'application (format **png**).
Après un clic droit sur le répertoire **res**, sélectionner **New/Image Asset** ; choisir le chemin d'accès à l'image choisie (champ **Path**). Inutile de changer le champ **Name**.

Cette image est mémorisée dans les répertoires **res/mipmap** dans plusieurs tailles pour s'adapter à différents supports.
Lancer l'exécution. L'icône s'affiche quand on met l'application en pause.

Etape 3 : Des boutons dans l'activité principale

Modifier le fichier **res/layout/activity_main.xml** de façon à supprimer l'affichage du texte initial. Placer trois boutons qui prennent la largeur de l'écran.
Cette modification peut se faire en écrivant du XML (onglet **Text**) ou bien en utilisant l'outil de design (onglet **Design**).



Ajouter une couleur de fond (attribut **background**) à chaque bouton, ainsi qu'un nom (attribut **id**).
Ces boutons sont pour l'instant inopérants.

Lorsqu'on colle une étiquette sur un bouton, IntelliJ suggère de mémoriser cette nouvelle chaîne dans **res/values/strings**, dans l'optique de faciliter la traduction de l'application. On trouve cette suggestion sous la forme d'une ampoule lorsqu'on promène le curseur sur la chaîne de caractères dans le texte **xml**.

Etape 4 : Un écouteur de bouton

L'écouteur du bouton Bonjour est créé dans la méthode **MainActivity.onCreate()**. On a accès à ce bouton, comme à toutes les vues du projet, par le biais de la fonction **findViewBy()**. Un clic sur ce bouton affiche un message court, appelé **Toast**. L'écouteur implémente l'interface **OnClickListener**. Penser à utiliser la complétion automatique pour trouver la fonction à implémenter.

```
Button bonjour = (Button) findViewById(R.id.bouton_bonjour);
bonjour.setOnClickListener((v) → {
        Toast.makeText(MainActivity.this, "Bonjour", Toast.LENGTH_SHORT).show() ;
});
```

Ces toasts sont bien pratiques pour afficher des messages courts à durée de vie limitée. Pensez-y...

Remarque : le code ci-dessus est écrit en Java 8. L'interface à implémenter n'a qu'une seule fonction, on a donc la possibilité d'écrire uniquement le texte de la fonction, sans s'embarrasser du profil que le compilateur est capable de retrouver tout seul.

Etape 5 : L'outil **LogCat**

Il est illusoire d'imaginer développer sans bug … Aussi, il faut un moyen de les traquer. L'habituel **System.out.println** ne convient plus dans ce contexte. Il faut utiliser l'outil **LogCat**, en utilisant une des fonctions statiques de la classe **Log**. Ces différentes fonctions permettent d'étiqueter les logs (information, warning, erreur, *etc.*). Par exemple, la fonction **i** affiche un message de type *information*.

Choisir un endroit quelconque dans le code Java où demander l'affichage d'un log. Vérifier son affichage dans la fenêtre **LogCat** ; choisir **Level Info** pour limiter l'affichage.

```
Log.i("MainActivity", "une info") ;
```

Pour que les logs s'affichent lors de l'exécution, il faut que la configuration d'exécution soit correctement positionnée : **Run configuration/Miscellaneous/Logcat**

Etape 6 : Un chronomètre

6.1 - Ajouter une nouvelle activité au projet (**File/New/Activity/Empty Activity**), portant le nom **ChronometreActivity**. Vérifier que cette activité est automatiquement ajoutée dans le fichier **AndroidManifest.xml**.

Compléter la méthode **MainActivity.onCreate()** de sorte que le clic sur le bouton **Chronometre** démarre cette nouvelle activité.

Pour cela, il faut créer un Intent qui relie l'activité courante **MainActivity** et l'activité à démarrer **ChronometreActivity**. L'intent créé est transmis à la fonction **startActivity**. Ici, l'activité **Chronometre** se fera sans retour.

```
Button chrono = (Button) findViewById(R.id.bouton_chrono);
chrono.setOnClickListener((v) → {
        // Intent avec activité actuelle et activité de destination
        Intent ic = new Intent(MainActivity.this, ChronometreActivity.class);
        // Puis on lance l'activité
        startActivity(ic);
});
```

6.2 - Compléter la méthode **ChronometreActivity.onCreate()** avec l'affichage d'un **Toast**, pour vérifier que la nouvelle activité est bien lancée.

6.3 - Dans l'activité **Chronometre**, ajouter deux boutons **Start/Stop**. Coller des images en background de ces boutons. Les images utilisées doivent être rangées dans **res/drawable** (par un simple glisser/déposer depuis l'arborescence de fichiers).

Un clic sur le premier bouton affiche l'heure dans un **Toast** et déclenche un chronomètre. Un clic sur le second bouton affiche le temps qui s'est écoulé dans un **Toast**. Seul un des deux boutons est cliquable.
Les écouteurs sont créés dans la méthode **onCreate()** de l'activité **Chronometre**.
Utiliser la classe **java.util.GregorianCalendar** ; son constructeur sans paramètre fournit la date courante.

Etape 7 : Prendre une photo

7.1 - On crée une nouvelle activité **Photo**. Dans le fichier **AndroidManifest.xml**, on déclare que la prise de photo est nécessaire.

```
<uses-feature android:name="android.hardware.camera"
    android:required="true" />
```

7.2 - Dans l'activité principale, un clic sur le bouton **Photo** permet de prendre une photo, sous réserve que le mobile dispose de cette fonctionnalité (test avec **resolveActivity**). Cette nouvelle activité renvoie un résultat, l'image prise (**startActivityForResult**). Le code passé en paramètre de la fonction **startActivityForResult** est celui de l'activité qui démarre (ici, l'attribut static entier **PHOTO**, fixé avec une valeur arbitraire).

```
Button photo = (Button) findViewById(R.id.bouton_photo);
photo.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent i = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
        if (i.resolveActivity(getPackageManager()) != null) {
            startActivityForResult(i, PHOTO);
        }
    }
});
```

A ce stade, le test du bouton est possible ; il reste à récupérer la photo au retour.

7.3 - Dans l'activité principale, les retours d'activités sont gérés par la fonction **onActivityResult()**. Cette fonction admet 3 paramètres : le code de l'activité qui se termine, un code de retour de l'activité, et le résultat.
Comme on a pu ouvrir plusieurs activités depuis la **MainActivity**, il faut tester le code de l'activité qui se termine, pour savoir quelle réaction adopter.
Le résultat renvoyé par une activité est dans un **Bundle**, que l'on consulte par la fonction **get**, sur la base d'une clé (ici, data). Ici, le résultat est de type **BitMap**.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == PHOTO && resultCode == RESULT_OK) {
        Bundle extras = data.getExtras();
        Bitmap imageBitmap = (Bitmap) extras.get("data");
    }
}
```

Pour s'assurer que la photo a bien été prise, afficher un **Toast** avec la hauteur de l'image. Par défaut, la photo est une miniature.

7.4 - Il reste à sauvegarder la photo. La sauvegarde peut se faire dans la mémoire (sauvegarde interne) ou sur une carte SD (sauvegarde externe). Dans le cas d'une sauvegarde interne, les données peuvent ou non être partagées en lecture/écriture par les autres applications ; dans le cas d'une sauvegarde externe, toutes les applications ont accès aux données.
Ici, on choisit une sauvegarde interne privée ; l'image est compressée au maximum (100).

```
FileOutputStream fos = openFileOutput("image.data", MODE_PRIVATE) ;
imageBitmap.compress(Bitmap.CompressFormat.PNG, 100, fos);
fos.flush();
fos.close();
```

7.5 - Ajouter un bouton dans l'activité principale ; un clic sur ce bouton permet d'afficher dans un composant de type **ImageView** la dernière photo prise.

```
FileInputStream fis = openFileInput("image.data") ;
Bitmap bm = BitmapFactory.decodeStream(fis);
ImageView iv = (ImageView)findViewById(R.id.imageView) ;
iv.setImageBitmap(bm);
```

Etape 8 : Une option de menu pour quitter l'activité

8.1 - Créer le répertoire de ressource **res/menu** et, dans ce répertoire, une ressource **menu_main.xml**. Ce menu est décrit par la balise **menu** ; chaque item de menu est décrit par la balise **item**.
Penser à donner un nom à ce composant.

8.2 - L'affichage du menu se programme dans la fonction **MainActivity.onCreateOptionsMenu**. Le composant graphique est créé à partir du fichier **xml** et attaché à l'activité. Le menu s'affiche en haut à droite sous 3 points.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_main, menu);
    return true;
}
```

8.3 - La réaction à la sélection d'item de menu se programme dans la fonction **MainActivity.onOptionsItemSelected**. Un test permet d'identifier l'item sélectionné. L'arrêt de l'activité est provoquée par un appel à la fonction **finish()**.

```java
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    if (id == R.id.action_quitter) {
        finish();
        return true;
    }
}
```

Une activité arrêtée subsiste en tâche de fond et pourra être reprise si l'utilisateur le demande. La seule façon de tuer définitivement l'application correspondante consiste à la faire glisser vers la droite de l'écran.

Etape 9 : A l'assaut du monde

La centralisation des ressources permet facilement d'internationaliser son application : par exemple, les chaînes de caractères utilisées sur les composants sont rangées dans **res/values/strings.xml**.

Ces chaînes sont utilisées par défaut. Si on souhaite que ces chaînes puissent être traduites en fonction de la langue du système (par exemple), il est nécessaire de créer des ressources adaptées.
Pour cela, ouvrir le fichier **strings.xml**. En haut à droite, choisir **Open editor**. Cet éditeur permet de créer de nouvelles ressources ; on choisit la langue en cliquant sur la planète. Chaque nouvelle langue donne lieu à la création d'un nouveau répertoire de ressources avec les chaînes traduites.
Ici, on se retrouve donc avec deux fichiers : **values/strings.xml** et **values-en/string.xml**, contenant tous deux la définition des chaînes de caractères, selon la langue.

Pour tester l'internationalisation de l'application, il suffit de changer la langue du système pour constater que les textes s'affichent dans la langue du système. Si la langue du système n'est pas référencée dans l'application, ce sont les chaînes par défaut qui sont utilisées.

Etape 10 : Un calcul long

10.1 - Dans l'activité principale, ajouter un bouton qui ouvre une nouvelle activité **LongActivity**. Cette nouvelle activité propose un champ de saisie d'un nombre entier et un bouton qui lance l'exécution d'une tâche longue dont la durée d'exécution dépend de l'entier saisi. Cette tâche longue peut être simplement un timer qui laisse le temps s'écouler, ou encore un calcul fictif long, peu importe.

10.2 - Pour éviter de bloquer le thread graphique pendant l'exécution de cette tâche longue, il est indispensable d'utiliser un nouveau thread. Cela permet également d'informer l'utilisateur de l'état d'avancement de la tâche.

Dans l'activité, ajouter une **ProgressBar** initialement invisible.
Ecrire une classe **TacheLongue** sous-classe de **AsyncTask<Params, Progress, Result>** (Doc) qui permet de créer un nouveau thread.

- Params, the type of the parameters sent to the task upon execution.
- Progress, the type of the progress units published during the background computation.
- Result, the type of the result of the background computation.

Si l'un des trois types n'est pas utile, on utilise **Void**.

Cette sous-classe doit définir les 4 fonctions ci-dessous.

- void onPreExecute(), invoked on the UI thread before the task is executed. This step is normally used to setup the task, for instance by showing a progress bar in the user interface.
- Result doInBackground(Params… p), invoked on the background thread immediately after onPreExecute() finishes executing. This step is used to perform background computation that can take a long time. The parameters of the asynchronous task are passed to this step. The result of the computation must be returned by this step and will be passed back to the last step. This step can also use publishProgress(Progress...) to publish one or more units of progress. These values are published on the UI thread, in the onProgressUpdate(Progress...) step.
- void onProgressUpdate(Progress… p), invoked on the UI thread after a call to publishProgress(Progress...). The timing of the execution is undefined. This method is used to display any form of progress in the user interface while the background computation is still executing. For instance, it can be used to animate a progress bar or show logs in a text field.
- void onPostExecute(Result r), invoked on the UI thread after the background computation finishes. The result of the background computation is passed to this step as a parameter.

Plus précisément,

- **TacheLongue** est sous-classe de **AsyncTask<Integer, Integer, Integer>**
- le constructeur admet en paramètre l'activité à l'origine de la création de la tâche et la mémorise pour pouvoir l'utiliser en cours de calcul ;
- la fonction **onPreExecute()** rend la **ProgressBar** visible ;
- la fonction **doInBackground(Integer... n)** effectue le calcul long sur la base de l'entier saisi et appelle **publishProgress** de temps en temps ;
- la fonction **onProgressUpdate(Integer... i)** affiche une information quelconque sur la progression de l'activité (soit sur un **Toast**, soit un autre autre composant, au choix) ;
- la fonction **onPostExecute()** rend la **ProgressBar** invisible.

Instancier cette sous-classe de **ASyncTask** lors du clic du bouton dans **CalculLong**. Exécuter la tâche créée, en lui fournissant l'entier saisi (fonction **execute**).

10.3 - L'utilisation de **ASyncTask** est potentiellement problématique, car si l'activité est arrêtée, la tâche continue de tourner, sauf si on l'arrête explicitement ; de plus, la tâche contient une référence (forte) à l'activité à l'origine de sa création et le ramasse-miettes ne pourra pas récupérer la mémoire, d'où de possibles fuites de mémoire. Pour résoudre le problème des fuites mémoire, il faut utiliser une **WeakReference** (ou référence faible). Le ramasse-miettes traite les références faibles comme des objets morts.

```
private final WeakReference<AppCompatActivity> myActivity;        // référence faible

public TacheLongue(AppCompatActivity a) { myActivity = new WeakReference<>(a); }
```

*A noter que la classe **ASyncTask** devient obsolète à partir de l'API 30, car son comportement est souvent mal géré et il est inconsistant sur différentes plate-formes ; des crash sont constatés. Il est largement conseillé de ne l'utiliser que pour des tâches potentiellement courtes, de l'ordre de quelques secondes.*

Etape 11 : Quel temps fait-il ?

On souhaite créer une nouvelle activité qui géolocalise le support mobile et affiche des informations locales relatives à la météo, comme la température, la description du temps (couvert, soleil, *etc.*) ou la vitesse du vent.
Comme la géolocalisation et la connexion Internet ne sont pas des tâches à réalisation immédiates, l'utilisation de **ASyncTask** est encore indispensable.
Pour réaliser cette activité, il faut apprendre à

- utiliser une API pour consulter la météo,
- utiliser la géolocalisation.

11.1 - Ajouter le nouveau bouton dans l'activité principale et la nouvelle activité **MeteoActivity**. Ajouter un bouton dans cette nouvelle activité.

11.2 - Compléter les permissions requises dans **AndroidManifest.xml** pour l'accès Internet et la géolocalisation.

```
<!-- Précision fine, avec matériel android.hardware.location.gps -->
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-feature android:name="android.hardware.location.gps"/>

<!-- Accès Internet -->
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
```

11.2 - Pour géolocaliser le support mobile, on doit avoir la permission de l'utilisateur. Si ce n'est pas le cas, on lui demande dans **MeteoActivity.onCreate()**. Une fois que c'est fait, on récupère le service qui gère la localisation.

```
// Vérifier qu'on a la permission de géolocalisation
// Sinon, on la demande à l'utilisateur
if ( ContextCompat.checkSelfPermission( context: this, Manifest.permission.ACCESS_FINE_LOCATION)
        != PackageManager.PERMISSION_GRANTED ) {
    ActivityCompat.requestPermissions( activity: this, new String[] {Manifest.permission.ACCESS_FINE_LOCATION  }, requestCode: 48 );
}

// Service en charge de la géolocalisation
final LocationManager locationManager = (LocationManager) this.getSystemService(LOCATION_SERVICE);
```

11.3 - Dans l'écouteur du bouton de l'activité **MeteoActivity**, demander au service de géolocalisation la longitude et la latitude courante et les afficher dans un **Toast** pour vérifier.

```
Location l = locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);
double latitude = l.getLatitude();
double longitude = l.getLongitude();
```

11.4 - Compléter l'écouteur avec la création d'une (nouvelle) tâche **TacheMeteo** de type **ASyncTask<String, Void, JSONObject**, sur le modèle de la classe **TacheLongue**. Les quatre fonctions se contentent d'afficher un **Log**.

11.5 - Pour consulter la météo, on utilise l'API OpenweatherMap (https://openweathermap.org/api). Il faut s'inscrire (Current Weather Data devrait suffire) et on récupère une clé à utiliser dans toutes les requêtes. Consulter la documentation (https://openweathermap.org/current) pour se familiariser avec l'écriture d'une requête qui contient la clé d'identification et les paramètres de la requête. Ecrire et tester une requête permettant de consulter la météo à Nancy ou à une latitude/longitude données.

11.6 - Compléter l'écouteur avec l'exécution de la tâche créée, en lui donnant en paramètre l'URL d'accès à l'API OpenweatherMap, incluant la latitude et la longitude fournies par le service de localisation et la clé d'accès. Ajouter un **Log** dans la fonction **doInBackGround** pour vérifier la bonne réception de l'URL dans la tâche.

11.7 - Il reste à finaliser la tâche :

- **onPreExecute** n'a rien à faire ;
- **doInBackground** se connecte à l'API passée en paramètre, récupère la réponse et la transforme en objet JSON ;
- **onPostExecute** décortique les données JSON pour y retrouver les informations utiles sur l'activité.

```
protected JSONObject doInBackground(String... urls) {
    URL url = null;
    HttpURLConnection urlConnection = null;
    JSONObject result = null;
    try {
        url = new URL(urls[0]);
        urlConnection = (HttpURLConnection) url.openConnection();
        InputStream in = new BufferedInputStream(urlConnection.getInputStream());
        result = readStream(in); // Lecture du stream et construction du JSON correspondant
    } catch (IOException | JSONException e) {
        e.printStackTrace();

    } finally {
        if (urlConnection != null)
            urlConnection.disconnect();
    }
    return result;
}

private JSONObject readStream(InputStream is) throws IOException, JSONException {
    StringBuilder sb = new StringBuilder();
    BufferedReader r = new BufferedReader(new InputStreamReader(is), sz: 1000);
    for (String line = r.readLine(); line != null; line = r.readLine()) {
        sb.append(line);
    }
    is.close();
    return new JSONObject(sb.toString());
}

protected void onPostExecute(JSONObject s) {
    try {
        double temp = s.getJSONObject("main").getDouble( name: "temp")-273.15;
        JSONObject desc = (JSONObject) s.getJSONArray( name: "weather").get(0);
        // A compléter
    } catch (JSONException e) {
        e.printStackTrace();
    }
}
```

Etape 12 : Mes contacts

On souhaite ajouter une nouvelle activité qui permet d'afficher les contacts (nom, téléphone). Là encore il faut gérer les permissions et utiliser un service qui permet de consulter les contacts enregistrés. C'est l'occasion d'utiliser le composant **RecyclerView** (une liste d'items).

La création de la nouvelle activité se fait comme toutes les autres. Voilà quelques pistes à suivre :

- Il est nécesssaire d'avoir la permission **READ_CONTACTS.**

- Le service **ContactsContrat** permet de consulter tous les numéros ; il faut itérer sur le résultat pour construire les contacts et les mémoriser dans un annuaire (une classe **Contact** + une classe **Annuaire**).

```
Cursor phones = getContentResolver().query(ContactsContract.CommonDataKinds.Phone.CONTENT_URI,  projection: null,
        selection: null,  selectionArgs: null,  sortOrder: null);
while (phones.moveToNext()) {
    String name = phones.getString(phones.getColumnIndex(ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME));
    String phoneNumber = phones.getString(phones.getColumnIndex(ContactsContract.CommonDataKinds.Phone.NUMBER));

    ……
```


- Le service **ContactsContrat** permet de consulter tous les numéros ; il faut itérer sur le résultat pour construire les contacts et les mémoriser dans un annuaire (une classe **Contact** + une classe **Annuaire**).

- Ce tuto est très utile pour apprendre à construire une **RecyclerView** :
  - Il faut compléter les dépendances dans **build.gradle.**

```
implementation 'androidx.recyclerview:recyclerview:1.1.0'
```

  - On a besoin d'une classe **ContactItem** qui contrôle un item de liste ; cette classe est sous-classe de la classe **RecyclerView.ViewHolder.**

  - On a besoin d'une classe adaptateur qui fait le lien entre le modèle (**Annuaire**) et le composant (**RecyclerView**) ; cette classe **AdaptateurContacts** est sous-classe de la classe **RecyclerView.Adapter<ContactItem>** et définit les fonctions
    - **AdapteurContacts(Annuaire an)**
    - **ContactItem onCreateViewHolder(ViewGroup parent, int viewType)**
    - **void onBindViewHolder(ContactItem contactItem, int position)**
    - **int getItemCount()**

    Le constructeur fait le lien avec le modèle de données. La première fonction crée une vue à partir de sa description XML ; la seconde met à jour un item donné ; la dernière donne le nombre total d'items.

    - Avec **ItemClickSupport.addTo** on peut ajouter un écouteur sur chaque item de la liste.

- Si le nombre de contacts est important, la création de l'annuaire peut être longue. Il est raisonnable de créer une nouvelle **ASyncTask** pour soulager le thread principal.

Etape 13 : Inspection de code

Le menu **Analyze** permet de lancer l'exécution de l'outil **lint**. Cet outil lance une analyse statique du code (la définition des classes, les dépendances entre les classes). Il permet aussi de faire le ménage dans le code.
A utiliser sans modération.

Quelques liens utiles

La documentation Android officielle
L'univers Android
Tâches asynchrones, requêtes réseau, **RecyclerView**
La gestion des permissions