

## Android - Semaine 5



Référence: Cours Mr Pierre Nerzie

Le cours de cette semaine concerne certains aspects de l'ergonomie d'une application Android.

- Menus et barre d'action
- Popup-up : messages et dialogues
- Activités et fragments

Et aussi, pour information :

- Préférences
- Bibliothèque support (androidx)

## Barre d'action et menus

# Barre d'action

La barre d'action contient l'icône d'application (1), quelques items de menu (2) et un bouton . pour avoir les autres menus (3).



# Réalisation d'un menu

Le principe général : un menu est une liste de d'items présentés dans la barre d'action. La sélection d'un item déclenche une *callback*.

Docs Android sur la [barre d'action](#) et sur [les menus](#)

Il faut définir :

- un fichier `res/menu/nom_du_menu.xml` qui est une sorte de layout spécialisé pour les menus,
- deux méthodes d'écouteur pour gérer les menus :
  - ajout du menu dans la barre,
  - activation de l'un des items.

# Spécification d'un menu

Créer `res/menu/nom_du_menu.xml` :



```
<menu xmlns:android="..." >
    <item android:id="@+id/menu_creer"
          android:icon="@drawable/ic_menu_creer"
          android:showAsAction="ifRoom"
          android:title="@string/menu_creer" />
    <item android:id="@+id/menu_chercher" ... />
    ...
</menu>
```

L'attribut `showAsAction` vaut "always", "ifRoom" ou "never" selon la visibilité qu'on souhaite dans la barre d'action. Cet attribut est à modifier en `app:showAsAction` si on utilise *androidx*.


# Icônes pour les menus

Android distribue gratuitement un grand jeu d'icônes pour les menus, dans les deux styles *MaterialDesign* : HoloDark et HoloLight.



Téléchargez l'[Action Bar Icon Pack](#) 📄 pour des icônes à mettre dans vos applications.

# Écouteur pour afficher le menu

Il faut programmer deux méthodes. L'une affiche le menu, l'autre réagit quand l'utilisateur sélectionne un item. Voici la première : 

```
@Override
public boolean onCreateOptionsMenu(Menu menu)
{
    // ajouter mes items de menu
    getMenuInflater().inflate(R.menu.nom_du_menu, menu);
    // ajouter les items du système s'il y en a
    return super.onCreateOptionsMenu(menu);
}
```

Cette méthode rajoute les items du menu défini dans le XML.

Un `MenuInflater` est un lecteur/traducteur de fichier XML en vues ; sa méthode `inflate` crée les vues.



## Écouteur pour afficher le menu, suite

On peut aussi ajouter des éléments de menu manuellement :



```
public static final int MENU_ITEM1 = 1;
...

@Override
public boolean onCreateOptionsMenu(Menu menu)
{
    ...
    // ajouter des items manuellement
    menu.add(Menu.NONE, MENU_ITEM1, ordre, titre).setIcon(image);
    ...
}
```

Cette fois, vous devrez choisir un identifiant pour les items. L'ordre indique la priorité de cet item ; mettre `Menu.NONE` s'il n'y en a pas.

# Réactions aux sélections d'items

Voici la *seconde callback*, c'est un aiguillage selon l'item choisi : 

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.menu_creer:
            ...
            return true;
        case MENU_ITEM1:
            ...
            return true;
        ...
        default: return super.onOptionsItemSelected(item);
    }
}
```

## Réactions aux sélections d'items, suite

Mais, dans les versions récentes d'Android Studio, les identifiants de menu ne sont plus des constantes et ne peuvent plus être utilisés dans un switch. On doit le transformer en conditionnelles en cascade :



```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    if (id == R.id.menu_creer) {
        ...
        return true;
    } else if (id == MENU_ITEM1) {
        ...
        return true;
    }
    ...
    } else return super.onOptionsItemSelected(item);
}
```

# Menus en cascade

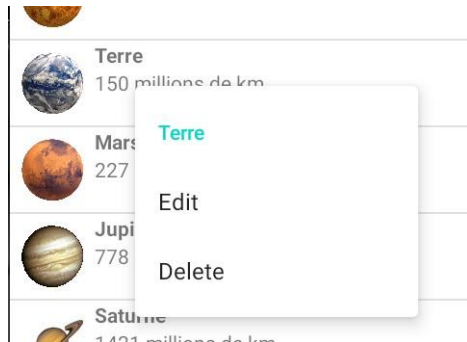
Définir deux niveaux quand la barre d'action est trop petite :



```
<menu xmlns:android="..." >
  <item android:id="@+id/menu_item1" ... />
  <item android:id="@+id/menu_item2" ... />
  <item android:id="@+id/menu_more"
        android:icon="@drawable/ic_action_overflow"
        android:showAsAction="always"
        android:title="@string/menu_more">
    <menu>
      <item android:id="@+id/menu_item3" ... />
      <item android:id="@+id/menu_item4" ... />
    </menu>
  </item>
</menu>
```

## Menus contextuels

# Menus contextuels



Ces menus apparaissent généralement lors un clic long sur un élément de liste. La classe `RecyclerView` ne possède rien pour afficher automatiquement des menus. Il faut soi-même programmer le nécessaire.

## Menus contextuels, suite

Voici les étapes :

- Le *View Holder* doit se déclarer en tant qu'écouteur pour des ouvertures de menu contextuel (clic long).
- La méthode déclenchée fait apparaître le menu contextuel.
- L'activité doit se déclarer en tant qu'écouteur pour les clics sur les éléments du menu contextuel.

Le souci principal, c'est qu'il n'y a pas de lien entre d'une part le *View Holder* qui observe le clic long sur un élément de la liste et fait afficher le menu contextuel, et d'autre part l'activité qui est réveillée quand l'utilisateur sélectionne un item du menu.

Il faut fournir la position de l'élément de la liste à l'activité, et on est obligé de bricoler.

## View Holder écouteur de menu

Il suffit d'ajouter ceci :



```
public class PlaneteViewHolder extends RecyclerView.ViewHolder
{
    private PlaneteBinding ui;

    public PlaneteViewHolder(@NonNull PlaneteBinding ui)
    {
        super(ui.getRoot());
        this.ui = ui;
        // pour faire apparaître le menu contextuel
        itemView.setOnCreateContextMenuListener(
            this::onCreateContextMenu);
    }
}
```

`itemView` est une variable de la classe `ViewHolder` qui est égale à `ui.getRoot()`.



## View Holder écouteur de menu, suite

Voici la *callback* d'un clic long sur un élément de la liste :



```
public static final int MENU_EDIT = 1;
public static final int MENU_DELETE = 2;

private void onCreateContextMenu(ContextMenu menu, View v,
    ContextMenu.ContextMenuInfo menuInfo)
{
    // position de l'élément
    int position = getAdapterPosition();
    // stocker la position dans l'ordre (3e paramètre)
    menu.add(Menu.NONE, MENU_EDIT, position, "Edit");
    menu.add(Menu.NONE, MENU_DELETE, position, "Delete");
    // titre du menu
    menu.setHeaderTitle(ui.nom.getText());
}
```

## View Holder écouteur de menu, suite et fin

Il y a une astuce, mais un peu faible : utiliser la propriété `order` des items de menu pour stocker la position de l'élément cliqué dans la liste. Cette propriété permet normalement de les classer pour les afficher dans un certain ordre, mais comme on ne s'en sert pas...

On est obligé de faire ainsi car il manque une propriété *custom* dans la classe `MenuItem`. Une meilleure solution serait de sous-classer cette classe avec la propriété qui nous manque, mais il faudrait modifier beaucoup plus de choses.

Une dernière remarque : il n'est pas possible d'afficher un icône à côté du titre d'item. C'est un choix délibéré dans Android.

# Écouteur dans l'activité

Enfin, il reste à rendre l'activité capable de recevoir les événements d'un clic sur un item du menu :



```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int position = item.getOrder();           // récup position
    Planete planete = liste.get(position);    // récup item
    switch (item.getItemId()) {
        case PlaneteViewHolder.MENU_EDIT:
            // TODO éditer planete (activité ou fragment...)
            return true;
        case PlaneteViewHolder.MENU_DELETE:
            // TODO supprimer planete (dialogue confirmation...)
            return true;
    }
    return false;    // menu inconnu, non traité ici
}
```

## Annonces et dialogues

## Annonces : *toasts*

Un « *toast* » est un message apparaissant en bas d'écran pendant un instant, par exemple pour confirmer la réalisation d'une action. Un *toast* n'affiche aucun bouton et n'est pas actif.



Voici comment l'afficher avec une ressource chaîne :

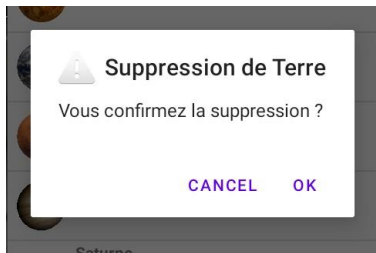


```
Toast.makeText(getApplicationContext(),  
    R.string.item_supprime, Toast.LENGTH_SHORT).show();
```

La durée d'affichage peut être allongée avec `LENGTH_LONG`.

# Dialogues

Un dialogue est une petite fenêtre qui apparaît au dessus d'un écran pour afficher ou demander quelque chose d'urgent à l'utilisateur, par exemple une confirmation.




Il existe plusieurs sortes de dialogues :

- Dialogues d'alerte
- Dialogues généraux

# Dialogue d'alerte

Un dialogue d'alerte [AlertDialog](#) affiche un texte et un à trois boutons au choix : ok, annuler, oui, non, aide...


Un dialogue d'alerte est construit à l'aide d'une classe nommée [AlertDialog.Builder](#). Le principe est de créer un *builder* et c'est lui qui crée le dialogue. Voici le début : 

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);  
builder.setTitle("Suppression de "+planete.getNom());  
builder.setIcon(android.R.drawable.ic_dialog_alert);  
builder.setMessage("Vous confirmez la suppression ?");
```

Ensuite, on rajoute les boutons et leurs écouteurs.

NB: utiliser [des ressources](#) pour les chaînes.

## Boutons et affichage d'un dialogue d'alerte

Le *builder* permet de rajouter toutes sortes de boutons : oui/non, ok/annuler. . . Cela se fait avec des fonctions comme celle-ci. On peut associer un écouteur (anonyme privé ou ...) ou aucun. 

```
// rajouter un bouton "oui" qui supprime vraiment
builder.setPositiveButton(android.R.string.ok,
    // écouteur écrit sous la forme d'une lambda
    (dialog, idbtn) -> supprimerVraiment(planete));
// rajouter un bouton "non" qui ne fait rien
builder.setNegativeButton(android.R.string.cancel, null);
```

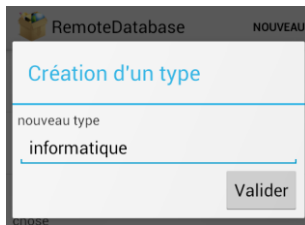
Enfin, on affiche le dialogue : 

```
// affichage du dialogue
builder.show();
```



# Dialogues personnalisés

Lorsqu'il faut demander une information plus complexe à l'utilisateur, mais sans que ça nécessite une activité à part entière, il faut faire appel à un [dialogue personnalisé](#).



# Création d'un dialogue

Il faut définir le layout du dialogue incluant tous les textes, sauf le titre, et au moins un bouton pour valider, mais pas pour annuler car on peut fermer le dialogue avec le bouton back.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="..." ...>
    <TextView android:id="@+id/titre" .../>
    <EditText android:id="@+id/libelle" .../>
    <Button android:id="@+id/valider" ... />
</LinearLayout>
```

Ensuite cela ressemble à ce qu'on fait dans `onCreate` d'une activité : installation du layout et des écouteurs pour les boutons.

# Affichage du dialogue



```
Dialog dialog = new Dialog(this);
DialogPersoBinding dialogUI =
    DialogPersoBinding.inflate(getLayoutInflater());
dialog.setContentView(dialogUI.getRoot());
dialog.setTitle("Création d'un type");
// bouton valider
dialogUI.valider.setOnClickListener(v -> {
    // récupérer et traiter les infos
    ...
    // ne surtout pas oublier de fermer le dialogue
    dialog.dismiss();
});
// afficher le dialogue
dialog.show();
```

## Fragments et activités

# Fragments

Un fragment est un sous-ensemble d'une interface d'application, par exemple :

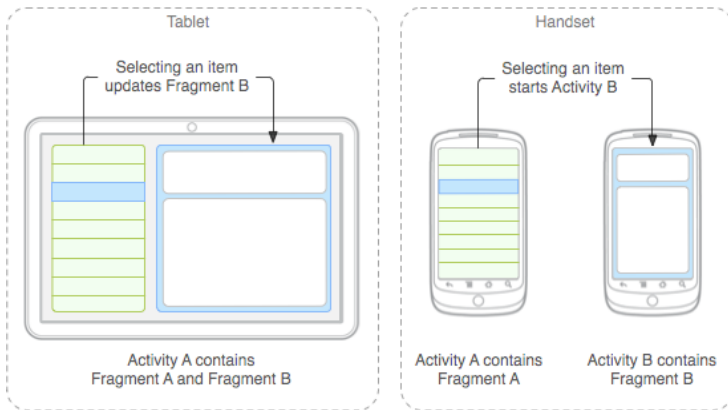
- liste d'items
- affichage des infos d'un item
- formulaire d'édition d'un item

Un *fragment* est une sorte de mini-activité restreinte à une seule chose : afficher une liste, afficher les informations d'un élément, etc.

Une activité peut être composée d'un ou plusieurs fragments, qui sont visibles ou non selon la géométrie du smartphone.

# Tablettes, smartphones...

Une interface devient plus souple avec des fragments. Selon la taille d'écran, on peut afficher une liste et les détails, ou séparer les deux.



# Différents types de fragments

Il existe différents types de fragments :

- [Fragment](#) superclasse, pour des fragments normaux.
- [DialogFragment](#) pour afficher un fragment dans une fenêtre flottante au dessus d'une activité.
- [PreferenceFragment](#) pour gérer les préférences.

En commun : il faut surcharger la méthode [onCreateView](#) qui met leur interface utilisateur en place. Si on utilise les View Bindings, `onCreateView` retourne simplement `ui.getRoot()`.


# Structure d'un fragment

Un fragment est une activité très simplifiée qui contient au moins un constructeur vide et `onCreateView` surchargée : 

```
public class InfosFragment extends Fragment {  
    public InfosFragment() {} // obligatoire  
  
    @Override  
    public View onCreateView(LayoutInflater inflater,  
        ViewGroup container, Bundle savedInstanceState) {  
        ui = PlaneteInfosBinding.inflate(inflater, container, false)  
  
        // écouteurs, adaptateur...  
  
        return ui.getRoot();  
    }  
}
```




## Structure d'un fragment, suite

Dans le cas du fragment Liste, c'est lui qui crée l'adaptateur : 

```
public class ListeFragment extends Fragment {  
    ...  
    private List<Planete> liste;  
    private PlaneteAdapter adapter;  
  
    @Override  
    public View onCreateView(LayoutInflater inflater, ...) {  
        ...  
        ... récupérer la liste (application ou BDD)  
        adapter = new PlaneteAdapter(liste);  
        ... layout manager, séparateur, écouteurs...  
  
        return ui.getRoot();  
    }  
}
```

# Menus de fragments

Un fragment peut définir un menu d'options dont les éléments sont intégrés à la barre d'action de l'activité. Seule la méthode de création du menu diffère, l'*inflater* arrive en paramètre : 

```
@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater mInf)
{
    mInf.inflate(R.menu.edit_fragment, menu);
    super.onCreateOptionsMenu(menu, mInf);
}
```


NB: dans la méthode `onCreateView` du fragment, il faut rajouter `setHasOptionsMenu(true)` ;

# Intégrer un fragment dans une activité

De lui-même, un fragment n'est pas capable de s'afficher. Il ne peut apparaître que dans le cadre d'une activité, comme une sorte de vue interne. On peut le faire de deux manières :

- statiquement : les fragments à afficher sont prévus dans le layout de l'activité. C'est le plus simple.
- dynamiquement : les fragments sont ajoutés, enlevés ou remplacés en cours de route selon les besoins (on ne verra pas).

# Fragments statiques dans une activité

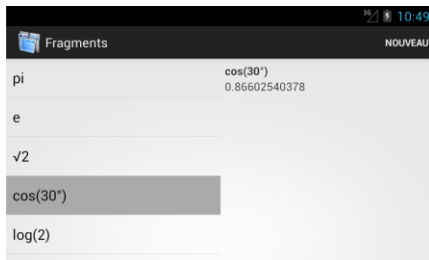
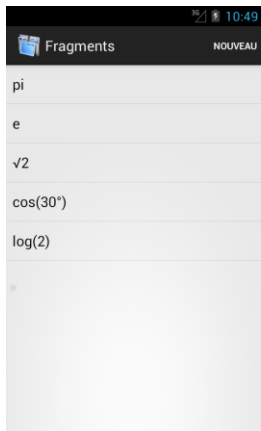
Dans ce cas, c'est le layout de l'activité qui inclut les fragments, p. ex. `res/layout/activity_main.xml`. Ils ne peuvent pas être modifiés ultérieurement. 

```
<LinearLayout android:orientation="horizontal" ... >
    <fragment    android:id="@+id/frag_liste"
        android:name="fr.iutlan.fragments.ListeFragment"
        ... />
    <fragment    android:id="@+id/frag_infos"
        android:name="fr.iutlan.fragments.InfosFragment"
        ... />
</LinearLayout>
```

Chaque fragment doit avoir un identifiant et un nom de classe complet avec tout le *package*. Ne pas oublier les attributs des tailles et éventuellement poids.

## Disposition selon la géométrie de l'écran

Le plus intéressant est de faire apparaître les fragments en fonction de la taille et l'orientation de l'écran (application « liste + infos »).



# Changer la disposition selon la géométrie

Pour cela, il suffit de définir deux layouts (des « variantes ») :

■ res/layout-port/activity\_main.xml en portrait :



```
<LinearLayout xmlns:android="..."
    android:orientation="horizontal" ... >
    <fragment android:id="@+id/frag_liste" ... />
</LinearLayout>
```

■ res/layout-land/activity\_main.xml en paysage :



```
<LinearLayout xmlns:android="..."
    android:orientation="horizontal" ... >
    <fragment android:id="@+id/frag_liste" ... />
    <fragment android:id="@+id/frag_infos" ... />
</LinearLayout>
```

## Deux dispositions possibles

Lorsque la tablette est verticale, le layout de `layout-port` est affiché et lorsqu'elle est horizontale, c'est celui de `layout-land`.

Pour savoir si le fragment `frag_infos` est affiché :



```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ...
    FragmentManager manager = getSupportFragmentManager();
    InfosFragment frag_infos = (InfosFragment)
        manager.findFragmentById(R.id.frag_infos);
    if (frag_infos != null) {
        // le fragment des informations est présent
        ...
    }
}
```

Notez que le fragment sera aussi mis en `@Nullable` dans le *view binding* de l'activité.

## Communication entre Activité et Fragments

Lorsque l'utilisateur clique sur un élément de la liste du fragment `frag_liste`, cela doit afficher ses informations :

- dans le fragment `frag_infos` s'il est présent,
- ou lancer une activité d'affichage séparée si le fragment n'est pas présent (layout vertical).

Cela implique plusieurs petites choses :

- L'écouteur des clics sur la liste doit être l'activité.
- L'activité doit déterminer si le fragment `frag_infos` est affiché :
  - s'il est visible, elle lui transmet l'item cliqué
  - sinon, elle lance une activité spécifique, `InfosActivity`.

Voici les étapes.



## Interface pour un écouteur

On reprend l'interface `OnItemClickListener` définie dans l'adaptateur, voir la fin du cours 4 :



```
public interface OnItemClickListener {  
    void onItemClick(int position);  
}
```

Ce sera l'activité principale qui sera cet écouteur dans l'adaptateur du fragment, grâce à :



```
@Override  
public View onCreateView(LayoutInflater inflater, ...) {  
    ...  
    adapter.setOnItemClickListener(  
        (PlaneteAdapter.OnItemClickListener) getActivity());  
    ...  
}
```

# Écouteur de l'activité

Voici maintenant l'écouteur de l'activité principale :



```
@Override public void onItemClick(int position)
{
    FragmentManager manager = getSupportFragmentManager();
    InfosFragment frag_infos = (InfosFragment)
        manager.findFragmentById(R.id.frag_infos);
    if (frag_infos != null && frag_infos.isVisible()) {
        // le fragment est présent, alors lui fournir la position
        frag_infos.setItemPosition(position);
    } else {
        // lancer InfosActivity pour afficher l'item
        Intent intent = new Intent(this, InfosActivity.class);
        intent.putExtra("position", position);
        startActivity(intent);
    }
}
```

## Relation entre deux classes à méditer

Une classe « active » capable d'avertir un écouteur d'un événement. Elle déclare une interface que doit implémenter l'écouteur.

```
public class Classe1 {  
    public interface OnEvenementListener {  
        public void onEvenement(int param);  
    }  
    private OnEvenementListener ecouteur = null;  
    public void setOnEvenementListener(  
        OnEvenementListener objet) {  
        ecouteur = objet;  
    }  
    private void traitementInterne() {  
        ...  
        if (ecouteur!=null) ecouteur.onEvenement(argument);  
    }  
}
```

## Relation entre deux classes à méditer, suite

Une 2<sup>e</sup> classe en tant qu'écouteur des événements d'un objet de Classe1, elle implémente l'interface et se déclare auprès de l'objet.

```
public class Classe2 implements Classe1.OnEvenementListener
{
    private Classe1 objet1;

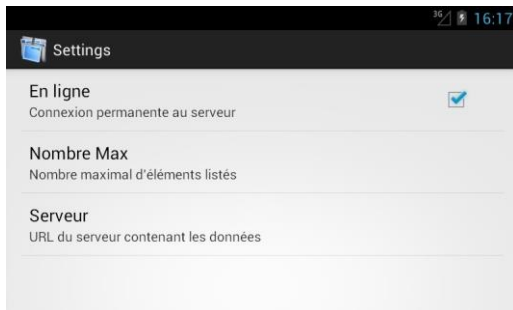
    public Classe2() {
        ...
        objet1.setOnEvenementListener(this);
    }

    public void onEvenement(int param) {
        ...
    }
}
```

## Préférences d'application

# Illustration

Les préférences mémorisent des choix de l'utilisateur entre deux exécutions de l'application.



# Présentation

Il y a deux concepts mis en jeu :

- Une activité pour afficher et modifier les préférences.
- Une sorte de base de données qui stocke les préférences,
  - booléens,
  - nombres : entiers, réels...
  - chaînes et ensembles de chaînes.

Chaque préférence possède un *identifiant*. C'est une chaîne comme "prefs\_nbmax". La base de données stocke une liste de couples (*identifiant*, *valeur*).

Voir la [documentation Android](#). Les choses changent beaucoup d'une version à l'autre d'API.

# Définition des préférences

D'abord, construire le fichier `res/xml/preferences.xml` :



```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="...">
    <CheckBoxPreference android:key="prefs_online"
        android:title="En ligne"
        android:summary="Connexion permanente au serveur"
        android:defaultValue="true" />
    <EditTextPreference android:key="prefs_nbmax"
        android:title="Nombre Max"
        android:summary="Nombre maximal d'éléments listés"
        android:inputType="number"
        android:numeric="integer"
        android:defaultValue="100" />
    ...
</PreferenceScreen>
```



# Explications

Ce fichier xml définit à la fois :

- Les préférences :
  - l'identifiant : `android:key`
  - le titre résumé : `android:title`
  - le sous-titre détaillé : `android:summary`
  - la valeur initiale : `android:defaultValue`
- La mise en page. C'est une sorte de layout contenant des cases à cocher, des zones de saisie. . . Il est possible de créer des pages de préférences en cascade comme par exemple, les préférences système.

Consulter [la doc](#) pour connaître tous les types de préférences.

NB: le résumé n'affiche malheureusement pas la valeur courante.

Consulter [stackoverflow](#) pour une proposition.

# Accès aux préférences

Les préférences sont gérées par une classe statique appelée `PreferenceManager`. On doit lui demander une instance de `SharedPreferences` qui représente la base et qui possède des *getters* pour chaque type de données.



```
// récupérer la base de données des préférences
SharedPreferences prefs = PreferenceManager
    .getDefaultSharedPreferences(getApplicationContext());


// récupérer une préférence booléenne
boolean online = prefs.getBoolean("prefs_online", true);
```

Les *getters* ont deux paramètres : l'identifiant de la préférence et la valeur par défaut.

# Préférences chaînes et nombres

Pour les chaînes, c'est `getString(identifiant, défaut)`.

```
String hostname = prefs.getString("prefs_hostname", "localhost");
```

Pour les entiers, il y a bug important (février 2015). La méthode `getInt` plante. Voir [stackoverflow](https://stackoverflow.com/questions/28812112/android-preferences-getint-crashes) pour une solution. Sinon, il faut passer par une conversion de chaîne en entier : 

```
int nbmax = prefs.getInt("prefs_nbmax", 99);    // PLANTE
int nbmax =
    Integer.parseInt(prefs.getString("prefs_nbmax", "99"));
```

# Modification des préférences par programme

Il est possible de modifier des préférences par programme, dans la base `SharedPreferences`, à l'aide d'un objet appelé *editor* qui possède des *setters*. Les modifications font partie d'une transaction comme avec une base de données.

Voici un exemple :



```
// début d'une transaction
SharedPreferences.Editor editor = prefs.edit();
// modifications
editor.putBoolean("prefs_online", false);
editor.putInt("prefs_nbmax", 20);
// fin de la transaction
editor.commit();
```

# Affichage des préférences

Il faut créer une activité toute simple :



```
public class PrefsActivity extends Activity {  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.prefs_activity);  
    }  
}
```

Le layout `prefs_activity.xml` contient seulement un fragment :



```
<fragment xmlns:android="..."  
    android:id="@+id/frag_prefs"  
    android:name="LE. PACKAGE. COMPLET. PrefsFragment"  
    ... />
```

Mettre le nom du package complet devant le nom du fragment.

# Fragment pour les préférences

Le fragment `PrefsFragment` hérite de [PreferenceFragment](#) : 

```
public class PrefsFragment extends PreferenceFragment {  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        // charger les préférences  
        addPreferencesFromResource(R.xml.preferences);  
        // mettre à jour les valeurs par défaut  
        PreferenceManager.setDefaultValues(  
            getActivity(), R.xml.preferences, false);  
    }  
}
```

C'est tout. Le reste est géré automatiquement par Android.

## [Bibliothèque support](#)

# Compatibilité des applications

Android est un système destiné à de très nombreux types de tablettes, téléphones, télévisions, voitures, lunettes, montres et autres. D'autre part, il évolue pour offrir de nouvelles possibilités. Cela pose deux types de problèmes :

- Compatibilité des matériels,
- Compatibilité des versions d'Android.

Sur le premier aspect, chaque constructeur est censé faire en sorte que son appareil réagisse conformément aux spécifications de Google. Ce n'est pas toujours le cas quand les spécifications sont trop vagues. Certains créent leur propre API, par exemple Samsung pour la caméra.



## Compatibilité des versions Android

Concernant l'évolution d'Android (deux versions du SDK par an, dont une majeure), un utilisateur qui ne change pas de téléphone à ce rythme est rapidement confronté à l'impossibilité d'utiliser des applications récentes.

Normalement, les téléphones devraient être mis à jour régulièrement, mais ce n'est quasiment jamais le cas.

Dans une application, le manifeste déclare la version nécessaire :

```
<uses-sdk android:minSdkVersion="20"  
          android:targetSdkVersion="32" />
```

Avec ce manifeste, si la tablette n'est pas au moins en API niveau 20, l'application ne sera pas installée. L'application est garantie pour bien fonctionner jusqu'à l'API 32 incluse.

# Bibliothèque support

Pour créer des applications fonctionnant sur de vieux téléphones et tablettes, Google propose une solution depuis 2011 : une API alternative, « *Android Support Library* ». Ce sont des classes similaires à celles de l'API normale, mais qui sont programmées pour fonctionner partout, quel que soit la version du système installé.

C'est grâce à des fichiers *jar* supplémentaires qui rajoutent les fonctionnalités manquantes.

C'est une approche intéressante qui compense l'absence de mise à jour des tablettes : au lieu de mettre à jour les appareils, Google met à jour la bibliothèque pour que les dispositifs les plus récents d'Android (ex: ActionBar, Fragments, etc.) fonctionnent sur les plus anciens appareils.

## Anciennes versions de l'Android Support Library

Il en existait plusieurs variantes, selon l'ancienneté qu'on visait. Le principe est celui de l'attribut `minSdkVersion`, la version de la bibliothèque : `v4`, `v7` ou `v11` désigne le niveau minimal exigé pour le matériel qu'on vise.

- `v4` : c'était la plus grosse API, elle permettait de faire tourner une application sur tous les appareils depuis Android 1.6. Par exemple, elle définit la classe `Fragment` utilisable sur ces téléphones. Elle contient même des classes qui ne sont pas dans l'API normale, telles que `ViewPager`.
- `v7-appcompat` : pour les tablettes depuis Android 2.1. Par exemple, elle définit l'`ActionBar`. Elle s'appuie sur la `v4`.
- Il y en a d'autres, plus spécifiques, `v8`, `v13`, `v17`.

# Une seule pour les gouverner toutes

Comme vous le constatez, il y avait une multitude d'API, pas vraiment cohérentes entre elles, et avec des contenus assez imprévisibles. Depuis juin 2018, une seule API remplace tout cet attirail : *androidx*.

Elle définit un seul espace de packages, *androidx.\** pour tout.

- Dans *app/build.gradle*, par exemple :

```
implementation "androidx.appcompat:appcompat:1.4.1"  
implementation "androidx.recyclerview:recyclerview:1.2.0"
```

# Une seule pour les gouverner toutes, fin

- Dans les layouts, par exemple :

```
<androidx.recyclerview.widget.RecyclerView .../>
```

- Dans les imports, par exemple :

```
import androidx.annotation.NonNull;  
import androidx.recyclerview.widget.RecyclerView;
```

Il reste à connaître les packages, mais on les trouve dans la documentation.

# Mode d'emploi

La première chose à faire est de définir le niveau de SDK minimal nécessaire, `minSdkVersion`, à mettre dans le `app/build.gradle` :

```
android {  
    compileSdkVersion 30  
  
    defaultConfig {  
        applicationId "mon.package"  
        minSdkVersion 20  
        targetSdkVersion 30  
    }  
}
```

## Mode d'emploi, suite

Ensuite, il faut ajouter les dépendances :

```
dependencies {  
    implementation fileTree(dir: 'libs', include: ['*.jar'])  
    ...  
    implementation "androidx.appcompat:appcompat:1.4.1"  
    implementation "androidx.recyclerview:recyclerview:1.2.0"  
    ...  
}
```

On rajoute les éléments nécessaires. Il faut aller voir la documentation de chaque chose employée pour savoir quelle dépendance rajouter, et vérifier son numéro de version pour avoir la dernière.

# Programmation

Enfin, il suffit de faire appel à ces classes pour travailler. Elles sont par exemple dans le package [androidx.fragment.app](#).

```
import androidx.fragment.app.FragmentActivity;
import androidx.recyclerview.widget.RecyclerView;

public class MainActivity extends FragmentActivity
...
```

Il y a quelques particularités, comme une classe `AppCompatActivity` qui est employée automatiquement à la place de `Button` dans les activités du type `AppCompatActivity`. Le mieux est d'étudier les documentations pour arriver à utiliser correctement tout cet ensemble.



## Il est temps de faire une pause

C'est fini pour cette semaine, rendez-vous la semaine prochaine pour un cours sur les adaptateurs de bases de données et les WebServices.