
Contenu

1. Plusieurs interfaces	1
2. Listes "adapters" interactives.....	5

1. Plusieurs interfaces

Pour le moment, une seule interface a été gérée. Il est temps de voir comment gérer plusieurs interfaces et passer de l'une à l'autre. Il existe plusieurs méthodes : nous allons en aborder une.

L'application va donc évoluer pour proposer une fonctionnalité supplémentaire : l'affichage de l'historique des mesures.

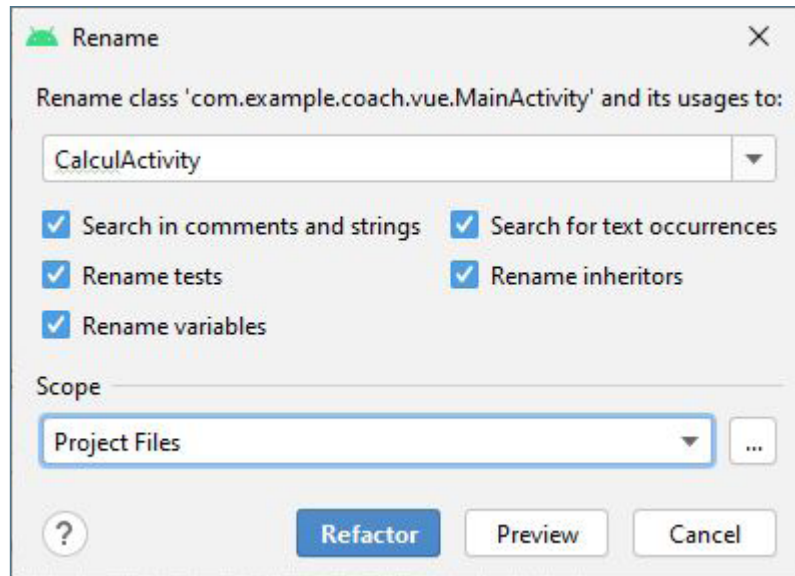
1A. Création de l'interface menu

L'interface principale ne doit plus amener directement sur la saisie des données mais doit permettre de choisir ce que l'on veut faire : saisir une nouvelle mesure (interface déjà faite) ou afficher la liste des mesures déjà enregistrées. Ce sera donc une interface de type menu (même si l'on ne va pas utiliser les fonctionnalités par défaut des menus Android).

1A1. Modification de MainActivity

MainActivity ne va plus être l'interface principale d'entrée dans l'application. Elle va devenir l'interface de saisie des informations pour le calcul de l'img.

Le but est donc dans un premier temps de changer le nom de MainActivity et de tout ce qui est lié à ce nom. Le faire à la main est trop long et risqué, car on peut oublier des modifications. Il faut utiliser les outils de refactoring de l'IDE. Faites un clic droit sur MainActivity (dans le package vue) et choisissez "Refactor > Rename...". Dans la fenêtre, mettez le nom CalculActivity et cochez toutes les cases :



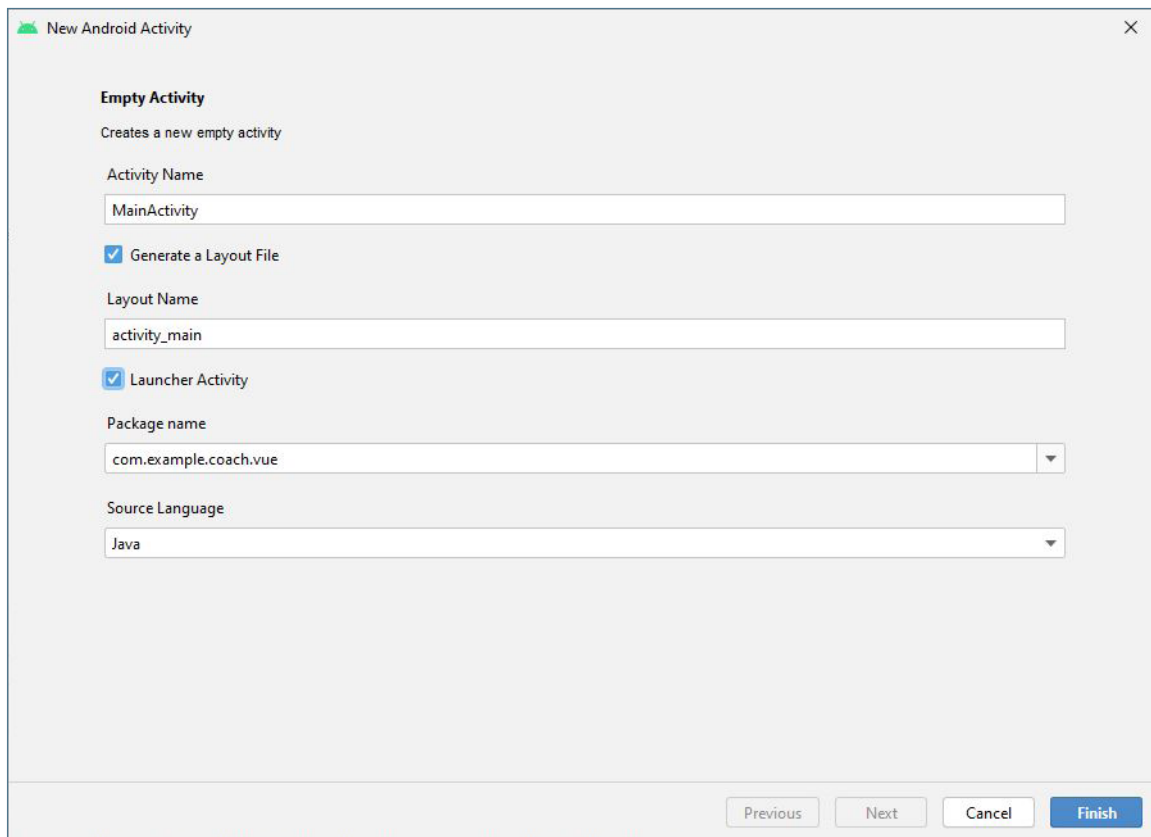
Cliquez sur Refactor. Contrôlez que tout a bien marché ne serait-ce qu'en allant voir, dans la classe Controle, méthode setProfil, si le transtypage a bien été remplacé par CalculActivity. Si vous remarquez que dans le package vue, il y a toujours marqué "MainActivity", c'est juste un bug de rafraîchissement : fermez le projet et ouvrez-le à nouveau. Il reste à modifier le nom de activity_main.xml (dans le dossier "res > layout") en utilisant aussi le refactoring, et en renommant en "activity_calcul" (cochez la case "Search in comments and strings"). Pourquoi cette modification n'a pas été faite automatiquement avec le refactoring précédent ? Parce qu'on a tout à fait le droit d'avoir le fichier xml d'un nom différent du fichier java qui y ait rattaché (d'ailleurs, même là, les noms sont différents).

Après cette transformation, lancez l'application pour voir si tout fonctionne correctement.

1A2. Ajout d'une Activity

Clic droit sur le package vue, sélectionnez "New > Activity > Empty Activity".

Cette fois le but est vraiment de créer la MainActivity qui contiendra le menu. Si les changements de noms précédents ont été correctement faits, la fenêtre devrait être déjà correctement remplie avec le nom "MainActivity" (cochez Launcher Activity pour que l'application démarre sur cette Activity) :



Contrôlez le contenu (attention au langage Java) et faites Finish.

Si vous rencontrez une erreur dans la nouvelle Activity, précisant que R est inconnu, faites l'import, soit avec alt+entrée sur R, soit en ajoutant à la main l'import correspondant :

```
import com.example.coach.R;
```

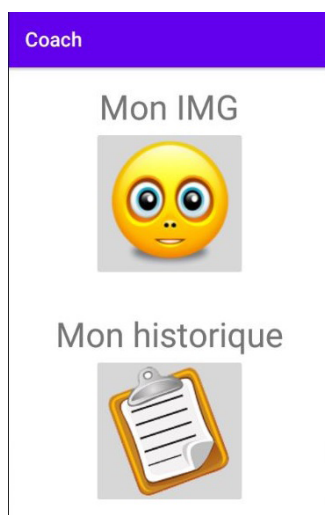
Allez dans AndroidManifest.xml. Le but est de faire en sorte que juste MainActivity soit l'activity de démarrage. Du coup, pour CalculActivity, enlevez les 4 lignes <intent-filter>.

Faites un test pour voir si vous atterrissez bien sur une Activity vierge.

1A3. Création du menu

Récupérez l'image historique dans les "sources", en la copiant dans "app > src > main > res > drawable" sur votre disque.

Voici l'interface que l'on veut créer :



Dans `activity_main.xml`, créez l'interface en utilisant des `ImageButton` pour les boutons du menu et en sélectionnant les bonnes images. Renommez les `ImageButton` en `btnMonIMG` et `btnMonHistorique` (pour positionner les objets graphiques, pensez à utiliser les layouts). Vous pouvez changer la taille d'écriture avec `textSize`. Ne soyez pas étonné si la taille des images vous paraît plus petite que la capture ci-dessus. Faites un test pour contrôler l'affichage : vous devriez obtenir l'affichage ci-dessus (avec l'AVD que l'on avait configuré).

1B. Lien vers une autre Activity

Le but va être de permettre de circuler entre les activités de l'application.

1B1. Passage du menu vers une autre activité

Dans le nouveau `MainActivity.java`, écrivez la méthode privée `creerMenu()`, qui ne retourne rien, ne reçoit rien, et qui doit être appelée à la fin de la méthode `onCreate`.

La méthode `creerMenu` appelle 2 fois la méthode `ecouteMenu` que vous allez écrire juste en dessous, et qui reçoit en paramètre un objet de type `ImageButton` et un objet final de type `Class` (que vous pouvez par exemple appeler "classe"). Lors des 2 appels de cette méthode, vous enverrez en premier paramètre l'objet `ImageButton` correspondant (faites en sorte de le récupérer avec `findViewById`) et en second paramètre l'objet `CalculActivity.class` (sachant que dès qu'on aura créé l'activité en rapport avec l'historique, on mettra son nom dans le 2^e appel, que vous pouvez d'ailleurs pour le moment mettre en commentaire).

Pourquoi avoir fait 2 méthodes : `creerMenu` et `ecouteMenu` ? Pour éviter d'écrire 2 fois le même code qui gère l'événement derrière chaque image. La méthode `ecouteMenu` va utiliser ses paramètres pour mettre une écoute sur la bonne image (premier paramètre) et, sur le clic de l'image, va ouvrir la bonne activité (second paramètre).

Il ne reste plus qu'à écrire la méthode `ecouteMenu`. Cette méthode doit capturer l'événement du clic (avec la méthode `setOnClickListener`) sur le bouton reçu en paramètre. Inspirez-vous du code que vous avez déjà géré pour capturer l'événement clic sur un bouton. Dans la méthode `onclick` qui est redéfinie à la volée, le but est d'accéder à la seconde activité. Pour cela, déclarez et créez un objet de type `Intent`. Le constructeur de la classe `Intent` attend en 1^{er} paramètre l'objet de l'activité actuelle (`MainActivity.this`) et en second paramètre l'objet de l'activité à ouvrir (prenez le paramètre que vous avez envoyé à `ecouteMenu`).

On va devoir passer plusieurs fois d'une activité à une autre. Pour éviter d'encombrer la mémoire, le mieux est de fermer l'activité actuelle avant d'ouvrir une autre. Cela se fait avec cette ligne de code (à écrire sur une seule ligne) :

```
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK | Intent.FLAG_ACTIVITY_CLEAR_TASK);
```

Maintenant que l'objet `intent` est déclaré et valorisé, envoyez-le en paramètre de la méthode `startActivity` (qui est une méthode à laquelle vous avez directement accès puisqu'elle est dans la classe mère).

Testez pour voir si le menu s'affiche et si, en cliquant sur le premier bouton, vous accédez bien à l'ancienne interface du calcul.

1B2. Retour vers le menu

Il faut aussi pouvoir revenir vers le menu. Vous pouvez utiliser le même principe. Pour cela, dans l'activité `Calcul`, ajoutez un `LinearLayout` (horizontal) tout en bas.

Ajoutez l'`ImageButton` accueil (après avoir récupéré l'image dans les sources), avec come id : `btnRetourDeCalcul` (car il faudra donner un id différent quand on mettra le même bouton pour revenir de la 3^e interface que l'on va bientôt créer, sachant que tous les id doivent être différents, même s'ils sont sur des interfaces différentes).

Pensez à mettre à 0 le `layout_weight` du nouveau layout, `layout_gravity` à `center_horizontal` et mettre son `layout_width` et `layout_height` à `wrap_content`. `Wrap_content` aussi pour le layout du bouton calculer (pour diminuer la zone en couleur).

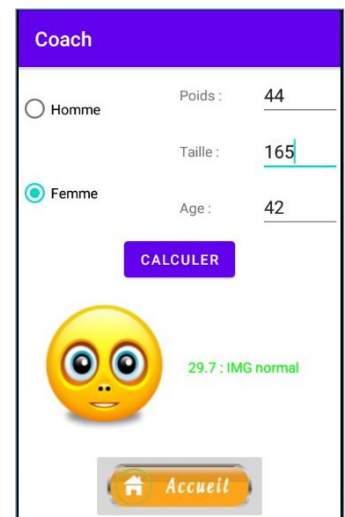
Vous devriez obtenir une interface qui ressemble à ceci :

Dans `CalculActivity`, derrière le clic sur l'image de retour à l'accueil, utilisez exactement le même principe pour revenir à l'accueil que ce que vous avez fait pour passer de l'accueil à cette activité (excepté qu'une seule méthode sera suffisante).

Testez. Vous devriez voir le menu apparaître, pouvoir aller sur "Mon IMG" et revenir au menu. Les anciennes informations du dernier profil ne s'affichent plus : c'est normal.

N'oubliez pas qu'à partir de maintenant, pour faire des tests sur un mobile, il faut installer les fichiers PHP et la base de données sur un serveur distant (voir les explications en fin de partie sur la base distante). Ceci dit, si vous installez l'APK sur votre mobile ou que vous branchez le mobile sur votre ordinateur pour un test direct, vous pourrez tout de même voir fonctionner le menu et le passage d'une activité à l'autre. En revanche vous ne pourrez pas enregistrer un profil.

Pensez à faire un "commit and push" pour l'enregistrement sur Bitbucket.



2. Listes "adapters" interactives

Le but est d'afficher une liste contenant l'historique des mesures précédemment faites. L'affichage de cette liste va se faire en suivant les règles des listes sur mobiles : les listes "adapters".

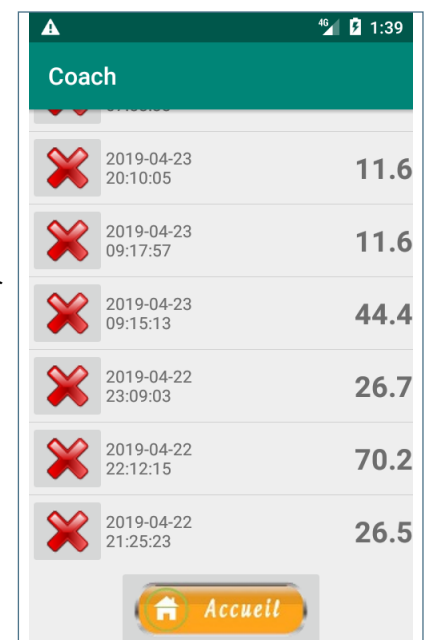
2A. Création d'une nouvelle activité

Créez la nouvelle activité `HistoActivity` dans le package `vue`.

Cette Activity va ressembler au final à la capture ci-contre.

Chaque ligne représente une mesure et va contenir des types d'objets différents par ligne (un bouton et des zones d'affichage de texte avec des formats différents), de plus il sera possible d'agir sur la ligne : par exemple, cliquer sur le bouton permettra de supprimer la ligne et donc de supprimer une mesure (dans la liste et dans la base distante), cliquer sur le reste de la ligne permettra d'aller dans `CalculActivity` et remplir les informations correspondant à la ligne sélectionnée.

Dans la nouvelle Activity, commencez par ajouter un `LinearLayout` vertical prenant toute la taille de la zone d'affichage, et dans ce layout, intégrez 2 `LinearLayout` (horizontal ou vertical, aucune importance) : le premier doit contenir un `ListView` (que vous nommerez `lstHisto`), le second doit contenir le bouton de retour au menu (`ImageButton` que vous nommerez `btnRetourDeHisto`). Distribuez les 2 layouts et modifiez les propriétés des layouts, comme vous l'avez fait pour `CalculActivity`, pour avoir le bouton en bas prenant peu de place (sans modifier la taille à la main).



	Date et heure	Valeur
X	2019-04-23 20:10:05	11.6
X	2019-04-23 09:17:57	11.6
X	2019-04-23 09:15:13	44.4
X	2019-04-22 23:09:03	26.7
X	2019-04-22 22:12:15	70.2
X	2019-04-22 21:25:23	26.5

Vous devriez obtenir ce visuel :

Remarquez l'ascenseur à droite : il est automatiquement configuré dans les ListView dans le cas où le contenu dépasse la taille de l'écran.

Faites aussi en sorte qu'à partir du menu, on puisse accéder à cette nouvelle Activity et que le retour vers le menu se fasse aussi. Pensez à créer une méthode init, dans la même logique que CalculActivity, qui pour le moment ne va contenir que l'appel de la méthode d'écoute du bouton du retour.

Même si pour le moment la liste est vide, testez pour voir si vous accédez bien à l'Activity à partir du menu, et si le retour marche aussi.

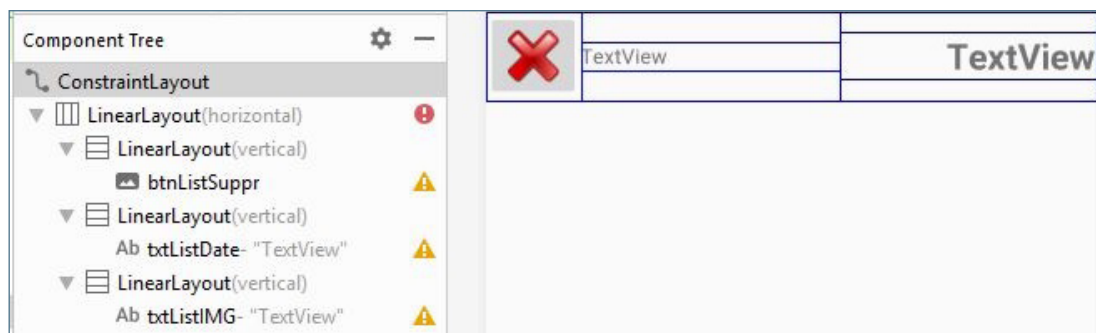


2B. Création du format de la ligne

Il faut construire le format de la ligne qui va servir de modèle pour remplir la liste. En effet, chaque ligne d'une liste "adapter" peut contenir n'importe quelle combinaison d'objets graphiques. Ici, on a besoin d'un bouton et de deux zones de texte.

Dans Android Studio, dans "res > layout", clic droit sur Layout et créez un "New > layout resource file". Donnez-lui le nom layout_liste_histo puis ok. Construisez le contenu en respectant les informations suivantes :

- ajouter un LinearLayout horizontal, et mettre le layout:height en match_parent (ce qui doit déjà être le cas) ;
- dans ce LinearLayout, ajouter et paramétrer 3 LinearLayout verticaux (après les avoir équilibrés en taille) avec layout:height en match_parent, et un gravity en center_vertical ;
- ajouter un TextView dans le layout central ;
- ajouter un TextView dans le layout de droite (il va contenir le calcul de l'IMG) et mettez en gras (bold dans textStyle), taille agrandie (par exemple 24sp dans textSize) et aligné à droite (gravity) ;
- ajouter dans le layout de gauche une ImageButton avec l'image de la croix (que vous aurez préalablement récupérée dans les sources) qui doit être de la taille de son contenu (layout:width et layout:height en wrap_content) ;
- mettre le layout de gauche (celui qui contient le bouton) en layout_width wrap_content et layout_weight à 0, pour qu'il soit de la largeur du bouton ;
- enfin, mettre en wrap_content le layout:height du premier grand layout (horizontal) pour qu'il s'adapte au contenu et soit donc moins haut verticalement ;
- donner les id comme montrés ci-dessous :



Vous allez maintenant pouvoir utiliser ce layout pour construire l'"adapter" qui va permettre de gérer chaque ligne de la liste, aussi bien au niveau de l'affichage qu'au niveau des éventuels événements (par exemple, sur le clic de la croix).

2C. Création d'une classe Adapter

Il faut maintenant créer cette liste multi-objets. Mais pour cela, il faut d'abord créer une classe particulière qui formate chaque ligne multi-objets de la liste. Cette notion est certainement une des plus complexes que vous avez pu voir depuis le début, au niveau de la programmation sous Android. Cependant, les listes "adapters" sont incontournables dans la programmation mobile sous Android.

2C1. Classe de type Adapter

Faites un clic droit sur le package vue et créez une Java Class simple en la nommant HistoListAdapter.

Pour que cette classe soit un Adapter, faites là hériter de la classe BaseAdapter. La signature provoque une erreur car la classe mère possède des méthodes abstraites, qui doivent donc être redéfinies. Cliquez sur la petite lampe rouge et choisissez "implement methods". La liste des méthodes qu'il faut implémenter apparaît dans une fenêtre : laissez-les toutes sélectionnées et cochez "Copy JavaDoc" pour avoir aussi les commentaires, puis faites ok. Quatre méthodes ont été insérées en @Override. En effet, l'Adapter doit permettre de retourner le nombre de lignes de la liste (getCount), l'item correspondant à un id (getItem), l'id correspondant à une position (getItemId) et la construction de la vue complète d'une ligne active contenant aussi les événements sur les objets de cette vue (getView) : ce sera la méthode la plus longue et la plus complexe.

2C2. Constructeur

Créez le constructeur qui reçoit en paramètre un contexte (de type Context) et une ArrayList d'objets de type Profil. L'ArrayList doit être stockée dans une propriété privée (lesProfils) à déclarer en tête de classe, et le contexte doit servir à valoriser la propriété inflater de type LayoutInflater (donc déclarez cette propriété en tête de classe). Pour valoriser cette propriété, vous devez lui affecter l'appel de la méthode statique from de la classe LayoutInflater. Cette méthode attend un contexte en paramètre. C'est tout pour le constructeur.

2C3. Méthodes simples

La méthode getCount doit retourner la taille de la liste lesProfils (utilisez la méthode size).

La méthode getItem qui reçoit une position en paramètre, doit retourner l'objet qui est dans lesProfils, à la position reçue en paramètre (utilisez la méthode get).

La méthode getItemId retourne tout simplement la position reçue en paramètre. Cette méthode semble ne servir à rien, mais elle pourrait être utilisée de façon plus exotique, pour retourner une ligne liée à une autre. On n'utilisera pas cette possibilité. Donc retournez juste le paramètre.

2C4. Formatage pour l'affichage

Avant de passer à la méthode la plus importante, nous allons écrire une méthode supplémentaire dans la classe MesOutils pour formater l'affichage à 2 chiffres après la virgule de l'IMG. Cette méthode va servir dans ce nouvel affichage. En fait, on s'est déjà occupé de formater correctement l'affichage de l'IMG, dans la méthode affichResult de CalculActivity. On avait utilisé la syntaxe suivante :

```
String.format("%.01f", img)
```

Vu qu'on en a besoin aussi ici, autant faire une méthode qui nous servira aux 2 endroits.

Donc, dans MesOutils, créez la méthode format2Decimal qui reçoit en paramètre un Float, et qui retourne un String formaté avec 2 chiffres après la virgule. Une fois cette méthode créée, utilisez-là dans affichResult de CalculActivity.

2C5. Lien avec les objets graphiques de la ligne

Il reste à remplir la méthode getView de HistoListAdapter. Mais pour cela, il faut définir une petite classe interne à notre classe, qui va permettre de donner la liste des types d'objets présents dans la ligne. Donc, directement dans la classe, comme si vous définissiez une nouvelle méthode, déclarez une classe privée ViewProperties qui contient juste la déclaration de 3 propriétés sans portée (pas private car on doit y accéder directement) : txtListDate et txtListIMG de type TextView, et btnListSuppr de type ImageButton. C'est tout.

2C6. Méthode getView

Revenons à la méthode getView. Supprimez le return actuel. La méthode reçoit 3 paramètres :

- position représente l'indice de la ligne d'affichage dans la liste ;
- convertView représente la ligne active d'affichage ;
- parent représente l'ensemble des lignes (on ne l'utilisera pas).



Remarque

Si les noms des paramètres sont différents, ce n'est pas grave. Soit vous les changez, soit vous en tenez compte dans la suite des explications.

L'idée globale de cette méthode est de construire la ligne d'affichage en affectant si nécessaire des informations dans les différents objets graphiques et en gérant éventuellement des événements sur les objets graphiques.

La petite classe ViewProperties contient les propriétés qui vont être liées aux objets graphiques de la ligne. Commencez par déclarer une variable viewProperties du type de cette classe.

Soit la vue (convertView) reçu en paramètre et représentant la ligne active, est null, dans ce cas il faut la construire, soit elle est remplie et on va juste récupérer son "tag" (étiquette : on verra plus loin à quoi cela correspond). Donc, commencez par tester si convertView est null. Si c'est le cas, créez viewProperties en lui affectant une instance de la classe correspondante.

La ligne (convertView) doit être construite à partir de layout_liste_histo. Ajoutez la ligne suivante :

```
convertView = inflater.inflate(R.layout.layout_liste_histo, null) ;
```

Chaque propriété de viewProperties doit être valorisée avec l'objet graphique correspondant, provenant de convertView. Par exemple, pour la première propriété, cela donne (sur une ligne) :

```
viewProperties.txtListDate = (TextView)convertView.findViewById(R.id.txtListDate) ;
```

Faites de même pour les deux autres propriétés.

Le but est maintenant d'enregistrer en étiquette (tag) l'objet viewProperties, qui contient les propriétés liées aux objets graphiques, dans convertView :

```
convertView.setTag(viewProperties) ;
```


Là vous comprenez mieux pourquoi on a regroupé les 3 propriétés dans une mini classe, afin d'utiliser un seul objet. Sinon, on n'aurait pas pu valoriser l'étiquette de convertView.

C'est fini pour la partie concernant la création de la ligne si elle n'existe pas. Maintenant, dans le else du test précédent (la ligne existe), récupérez, avec la méthode getTag, l'étiquette précédemment enregistrée, et utilisez là pour valoriser viewProperties, après l'avoir transtypé en ViewProperties.

Après la fermeture du test, la ligne existe (elle vient d'être créée ou elle a été récupérée) et le tag contient viewProperties (et vice versa). Pour rappel, viewProperties contient les 3 objets graphiques de la ligne.

Vous allez pouvoir maintenant affecter la date de mesure et la valeur de l'IMG aux objets graphiques correspondants. Pour cela, il faut utiliser la propriété setText sur les propriétés txtListDate et txtListIMG de l'objet viewProperties. En paramètre du setText, il faut récupérer les propriétés correspondantes de l'objet de type Profil qui se trouve à l'indice "position" dans la collection lesProfils (c'est la méthode get sur lesProfils, en précisant l'indice, qui permet de récupérer un objet précis de la collection). Attention, la date de mesure doit être convertie en String et l'IMG doit être formaté avec 2 chiffres après la virgule : vous avez 2 méthodes qui font ces conversions, dans la classe MesOutils.

Il ne reste plus qu'à retourner convertView.

La classe HistoListAdapter est pour le moment terminée. On y reviendra pour ajouter le code correspondant au clic sur le bouton de suppression d'une ligne.

Cette classe, vous pourrez la récupérer, en l'adaptant, dans toutes vos applications qui utiliseront des listes adapters.

2D. Récupération de tous les profils de la base distante

Avant de remplir la liste, et donc de pouvoir tester, il faut récupérer tous les profils de la base distante. Officiellement vous devriez savoir-faire. Cela représente tout de même beaucoup de modifications. Si vous n'y arrivez pas, voici la trame à suivre :

- modifiez le code de la page PHP pour récupérer tous les profils : le code est proche de la récupération du dernier, excepté qu'il faut boucler sur le curseur et, à chaque tour de boucle, insérer la ligne dans une case d'un tableau (au lieu de faire directement print) :

```
$resultat[] = $ligne;
```

Il suffit alors, après la boucle, de retourner le tableau après l'avoir encodé en json :

```
print(json_encode($resultat));
```

N'oubliez pas, avant la boucle, de faire un print("tous%"), si vous avez nommé l'opération "tous" ;

- dans le contrôleur, déclarez et créez la propriété lesProfils (pas en static) qui est une ArrayList d'objets de type Profil, et créez le getter et setter correspondants ;
- dans le contrôleur, méthode getInstance, faites appel au serveur distant en envoyant l'opération "tous" à la place de "dernier" ;
- dans le contrôleur, modifiez la méthode creerProfil qui va créer en local un profil (déjà fait), l'ajouter dans la liste et continuer à l'envoyer au serveur. Pour les getters sur le message et l'img, retournez les informations du dernier ajouté dans lesProfils (ou 0 pour l'IMG, "" pour le message, si la liste est vide, ce qui normalement ne devrait pas arriver). S'il reste des méthodes avec des erreurs (recupSerialize, setProfil) mettez ces méthodes en commentaire ou supprimez-les car elles ne serviront plus ;
- du coup, dans CalculActivity, puisqu'on n'affiche plus les informations du dernier, mettez en commentaire ou supprimez l'appel de la méthode recupProfil (si ce n'est pas déjà fait) ;
- dans AccesDistant, méthode processFinish, le test n'est plus sur "dernier" mais sur "tous". Dans ce test, vous ne recevez pas un objet mais un ensemble d'objets (tous les profils) dans un tableau json. Donc, commencez par convertir message[1] en JSONArray. Vous pourrez alors boucler sur les

cases du JSONArray pour convertir chaque case en JSONObject (comme vous l'aviez déjà fait mais attention, les cases contiennent des Object et le constructeur de JSONObject attend un String, donc concaténez "" juste devant l'objet ou ajoutez .toString()) et ainsi remplir une collection (ArrayList) d'objets de type Profil (à déclarer et instancier avant la boucle) pour finalement, après la boucle, l'envoyer au contrôleur avec la méthode setLesProfils.

Si vous avez correctement effectué les modifications, vous ne devriez plus avoir d'erreur dans le code (ce qui ne veut cependant pas dire que tout va fonctionner correctement). On ne peut malheureusement toujours pas tester.

2E. Création d'une liste liée à un Adapter

Dans le code de HistoActivity, créez la méthode privée creerListe qui ne reçoit aucun paramètre et ne retourne rien. Cette méthode doit être appelée en fin de init.

La méthode creerListe doit déclarer un objet liste de type ArrayList qui contiendra des objets de type Profil. Pour remplir cette liste, utilisez le getter sur la liste lesProfils qui est dans le controleur (ce qui suppose que vous devez déclarer une propriété de type Controle et la valoriser dans la méthode init, avec la récupération de l'instance de Controle).

La suite du code de la méthode creerListe ne doit se faire que si la liste récupérée n'est pas null. Faites le test en conséquence.

Dans le cas où la liste n'est pas null, déclarez un objet listView de type ListView et récupérez votre listView graphique (à partir de son id : lstHisto).

Cette liste doit contenir des lignes contenant elles-mêmes des objets différents (bouton, texte...). C'est donc là que l'on va utiliser l'Adapter. Commencez par déclarer un objet adapter du type de votre classe HistoListAdapter, et affectez à cet objet une instance de la classe (attention aux paramètres du constructeur : envoyez en premier paramètre le contexte actuel HistoActivity.this, et en second paramètre l'ArrayList que vous venez de créer). Enfin sur l'objet listView, appliquez la méthode setAdapter en lui envoyant en paramètre l'adapter que vous venez de créer.

La méthode creerListe est terminée.

Il est temps de faire un test. Contrôlez que vous avez bien des profils dans MySQL (sinon, ajoutez-en quelques-uns). Lancez l'application, allez directement dans l'historique : normalement rien ne s'affiche. Revenez au menu et retournez à l'historique : cette fois tout s'affiche. Si ce n'est pas le cas, revenez en arrière : vous avez fait une erreur quelque part (utilisez le débogueur, contrôlez les retour du serveur dans le logcat...). Quelques pistes pour rechercher l'erreur : commencez déjà par contrôler que vous recevez bien les profils provenant du serveur (normalement dans la console, vous devez avoir l'affichage "tous%" suivi des profils). Ensuite, dans AccesDistant, contrôlez que vous remplissez bien la collection de profils (il suffit de faire un Log.d sur la taille de la collection, une fois remplie). Vérifiez que lesProfils sont aussi remplis dans la classe Contrôle. Enfin, vérifiez dans HistoActivity, que vous appelez bien la méthode creerListe et la liste contient quelque chose.

En revanche, si cela fonctionne comme je viens de l'expliquer, c'est normal : le temps que le serveur réponde, vous avez déjà affiché la liste, du coup en sortant et retournant sur la liste, vous pouvez enfin la voir.

Les profils sont récupérés sur le serveur au moment de la création de l'instance du contrôleur. Pour le moment, cette instance n'est créée que si on va dans le calcul ou dans l'historique. Il faut donc créer cette instance dès le MainActivity : faites la correction nécessaire.

Refaites le test : normalement la liste doit se remplir du premier coup.

Essayez aussi d'ajouter de saisir et calculer un nouvel IMG puis retournez dans l'historique pour voir s'il apparaît.

En revanche, vous remarquez que la liste n'est pas triée sur la date. Ce serait bien qu'elle affiche les profils du plus récent au plus ancien. Pour cela, allez dans la classe HistoActivity, méthode creerListe, et juste après avoir rempli la variable liste et si la liste n'est pas vide,, écrivez la ligne suivante :

```
Collections.sort(liste, Collections.<Profil>reverseOrder());
```

Il est possible de demander un tri d'une liste avec la méthode statique sort de la classe Collections. Cette méthode reçoit soit un seul paramètre (la liste à trier), soit 2 paramètres (le second permettant de préciser le sens du tri : ici on le veut dans le sens inverse).

Il est possible que la ligne de code soit soulignée car l'ordinateur n'arrive pas à comparer des objets de type Profil. Si la ligne n'est pas soulignée en rouge, vous obtiendrez de toute façon une erreur en lançant l'application et en allant dans l'historique. Pour que l'ordinateur arrive à comparer des objets de type Profil, il faut qu'il sache sur quoi il doit comparer. Pour cela, il faut d'abord que la classe Profil implémente l'interface Comparable. Donc, allez dans la classe profil et ajoutez cette interface. Vous êtes alors invité à insérer une méthode : compareTo. Faites l'insertion automatique avec la lampe rouge.

Dans la méthode compareTo, qui reçoit en paramètre un objet o de type Object, vous allez comparer la dateMesure actuelle avec la dateMesure de l'objet reçu en paramètre, qu'il faudra bien sûr transtyper (la comparaison va se faire aussi avec compareTo car la classe Date implémente cette méthode). Voici la ligne de code à écrire :

```
return dateMesure.compareTo(((Profil)o).getDateMesure());
```

La méthode compareTo renvoie un nombre négatif si la première date est plus petite que la seconde, renvoie 0 si les deux dates sont identiques, renvoie un nombre positif si la première date est plus grande que la seconde.

Refaites un test pour voir si la liste s'affiche dans le bon ordre.

Bien sûr le bouton de suppression n'est pas encore actif. On va s'en occuper.

2F. Possibilité de supprimer des lignes de la liste

Le but est, sur le clic d'un bouton de suppression, de supprimer la ligne mais aussi la mesure dans la base distante.

Dans la classe HistoListAdapter, méthode getView, juste avant le return final, on va insérer le code nécessaire.

Avant tout, il faut mémoriser l'indice de ligne pour pouvoir le retrouver dans l'événement onClick. Pour cela, on va encore une fois utiliser le tag, décidément bien pratique pour mémoriser des informations : avec la méthode setTag sur btnListSuppr (accessible à partir de viewProperties), mémorisez la variable position.

Comme vous l'avez déjà fait plusieurs fois, écrivez le code qui permet de gérer le clic sur le bouton btnListSuppr (accessible dans viewProperties).

Dans la méthode onClick, commencez par récupérer, dans une variable locale indice de type int, le tag de la ligne concernée : pour récupérer ce tag, vous devez appliquer la méthode getTag() sur l'objet v (v est l'objet de type View reçu en paramètre de la méthode onClick, et qui représente la ligne concernée par le clic). Vous comprenez que la méthode getTag() permet de récupérer ce que vous aviez mémorisé avec la méthode setTag un peu plus haut. Pensez à caster en int comme cela est demandé.

Ensuite vous devez accéder au contrôleur pour l'informer de la suppression du profil concerné. Il faut donc créer une variable locale controle dans laquelle vous affectez l'instance unique du contrôleur (toujours avec la méthode getInstance et en mettant null en paramètre).

Vous allez pouvoir alors solliciter une méthode du contrôleur qui va gérer la suppression. Cette méthode n'existe pas encore. Vous allez donc devoir la créer.

Donc, dans le contrôleur, créez la méthode `delProfil` qui ne retourne rien, qui reçoit un objet de type `Profil` et qui doit envoyer au serveur distant la demande de suppression : inspirez-vous de ce que vous avez fait pour enregistrer un profil dans la base distante, cependant, cette fois, même si vous envoyez au serveur un profil complet (au format json), il suffira de récupérer la case contenant `datemesure`, pour écrire la requête delete (car `datemesure` est la clé primaire). La méthode `delProfil` doit aussi supprimer le profil dans la collection `lesProfils` (avec `remove` et en envoyant le profil reçu en paramètre).

Enfin, de retour dans `HistoListAdapter`, méthode `onClick`, appelez la méthode `delProfil` sur l'objet `controle` en envoyant en paramètre le profil qui se trouve au bon indice, dans la collection `lesProfils`.

Il ne reste plus qu'à demander que la liste "adapter" soit rafraîchie pour que l'on voit en direct la suppression. Pour cela, ajoutez l'instruction :

```
notifyDataSetChanged() ;
```

Testez pour voir si tout fonctionne correctement. Pensez à contrôler que la suppression se fait dans la base distante.

2G. Possibilité d'afficher un profil dans `CalculActivity`

Le but est de gérer un événement sur le reste de la ligne de sorte qu'en cliquant sur la ligne, l'activité `Calcul` s'ouvre avec les informations du profil de la ligne sélectionnée. Pour cela, il faut gérer en fait 2 événements : sur la date et sur l'img.

Dans `HistoListAdapter`, reprenez les mêmes lignes de code que précédemment, sur le clic sur la croix, et copiez-les juste en dessous évidemment en faisant les corrections nécessaires pour que ce soit l'événement sur la date ou l'IMG (sans oublier, avant l'événement, la ligne de code qui permet de mémoriser le tag).

Dans chaque `onClick`, vous allez avoir le même code :

- la récupération du tag dans `position` (comme vous l'avez déjà fait) ;
- l'appel d'une méthode non encore écrite, `afficheProfil`, dans `HistoActivity`, en lui envoyant en paramètre le profil actuel (celui qui se trouve à la position récupérée par le tag, dans `lesProfils`).

Pour appeler cette méthode, il faut pouvoir accéder à `HistoActivity`. Mais pour le moment, le constructeur de `HistoListAdapter`, qui récupère bien un contexte, ne mémorise pas le contexte dans une propriété. Faites donc en sorte de mémoriser le contexte dans une propriété. Bien sûr au moment de l'appel de `afficheProfil`, il faudra transtyper `contexte` en un objet de type `HistoActivity`.

Dans la classe `HistoActivity`, écrivez la méthode `afficheProfil` qui ne retourne rien et qui reçoit en paramètre un profil. Elle doit d'abord appeler `setProfil` de la classe `controle` pour valoriser le profil, puis elle doit demander d'ouvrir `CalculActivity`. Pourquoi on n'a pas mis cette méthode directement dans `HistoListAdapter` ? Parce que seule une `Activity` peut ouvrir une autre `Activity`.

Dans la classe `Controle`, il faut que la propriété `profil` ne soit valorisée que par l'appel de `setProfil`, provenant de `HistoActivity`. Donc, dans `creerProfil`, ne valorisez pas la propriété `profil` mais déclarez en local un `unProfil` de type `Profil` pour l'utiliser dans toute cette méthode mais uniquement dans cette méthode.

Dans la méthode `setProfil`, si ce n'est pas déjà fait, mettez en commentaire ou supprimez la ligne qui appelle `recupProfil` dans `CalculActivity`. La méthode `setProfil` ne doit que valoriser le profil.

Dans la classe `CalculActivity`, méthode `init()`, enlevez le commentaire devant l'appel de `recupProfil()` (ou rajoutez l'appel, si vous l'aviez supprimé). Enfin, dans la méthode `recupProfil`, tout à la fin, le but est de vider le profil du `controle` pour pas qu'il s'affiche à chaque fois qu'on retourne sur `CalculActivity`. Pour cela, appelez la méthode `setProfil` sur l'objet `controle`, et envoyez `null` en paramètre.

Si ce n'est pas déjà fait, supprimer la ligne de code qui déclenche le clic sur le bouton calcul : cela évitera de mémoriser systématiquement l'ancien profil (qui serait alors en double). Car le but de récupérer le profil, c'est éventuellement de faire une petite modification avant de recalculer.

Testez pour voir si tout fonctionne correctement. Allez dans la liste, cliquez sur une ligne : normalement CalculActivity s'ouvre en affichant les informations du profil. Contrôlez que c'est le bon profil en faisant "calcul" et en retournant dans la liste pour comparer les IMG. Essayez de revenir au menu et d'aller à nouveau dans CalculActivity, cette fois sans passer par la liste : normalement les informations ne sont plus présentes.

N'oubliez pas que pour faire des tests sur un mobile, il faut installer les fichiers PHP et la base de données sur un serveur distant (voir les explications en fin de partie sur la base distante).

Pensez à faire un "commit and push" pour l'enregistrement sur Bitbucket.

À travers la construction de cette application, vous avez eu un premier aperçu des possibilités de la programmation sous Android. Il est temps, par vous-même, de découvrir d'autres fonctionnalités importantes comme l'exploitation des capteurs du smartphone (appareil photo, gyroscope, accéléromètre...), les données incluses dans le smartphone (photos, agenda...) et les web services accessibles à partir du smartphone (géolocalisation...). Beaucoup de choses à découvrir.



Les cours du CNED sont strictement réservés à l'usage privé de leurs destinataires et ne sont pas destinés à une utilisation collective. Les personnes qui s'en serviraient pour d'autres usages, qui en feraient une reproduction intégrale ou partielle, une traduction sans le consentement du CNED, s'exposeraient à des poursuites judiciaires et aux sanctions pénales prévues par le Code de la propriété intellectuelle. Les reproductions par reprographie de livres et de périodiques protégés contenues dans cet ouvrage sont effectuées par le CNED avec l'autorisation du Centre français d'exploitation du droit de copie (20, rue des Grands-Augustins, 75006 Paris).

CNED, BP 60200, 86980 Futuroscope Chasseneuil Cedex, France

© CNED 2021

87D22TDWB1D21

