ProgIHM – TP4

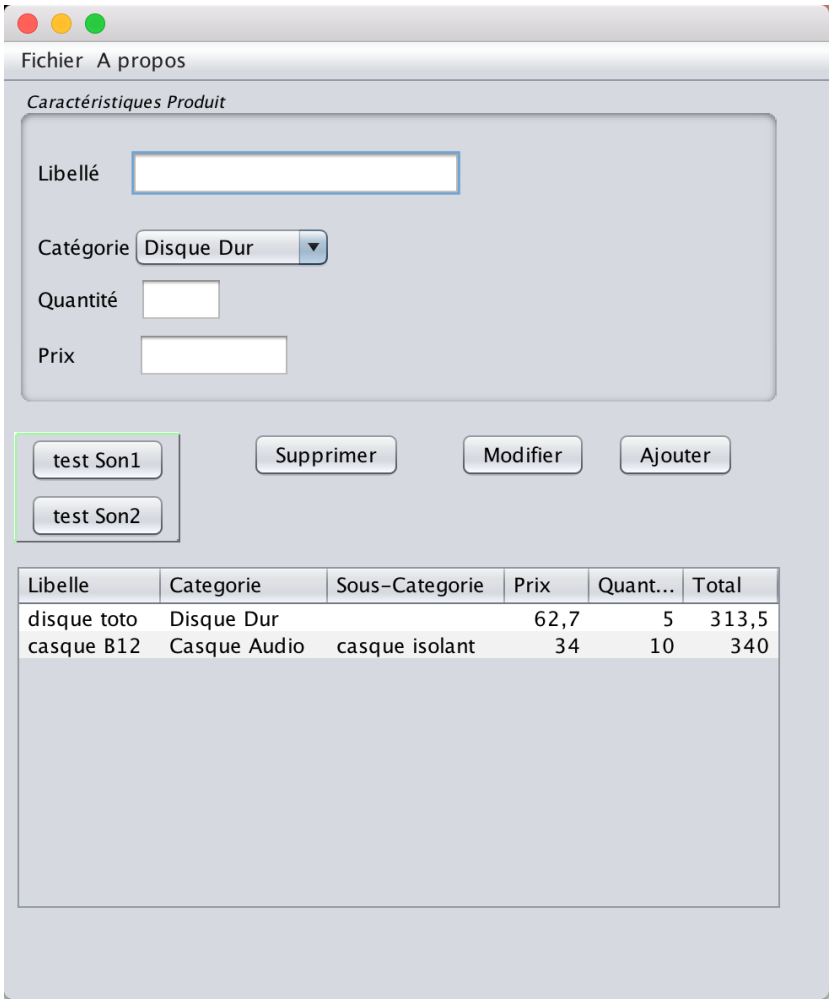
JTable, JList, Barre d'outils

Durée : 4h

L'objectif du TP est de **poursuivre notre interface Gestion de Produits.** On imagine qu'il s'agit d'une application du service Approvisionnement, pour des personnels d'une entreprise, qui permet de faire l'inventaire. On veut pouvoir saisir et stocker des produits, présents dans l'entreprise.

Dans ce TP on va aussi apprendre à *débugguer* son programme. Pour les interfaces : https://netbeans.apache.org/kb/docs/java/debug-visual.html

---



1- On va utiliser une **JTable** tabProduits pour afficher tous les produits.

a. La table comportera 6 colonnes pour afficher les informations : Libellé, Catégorie, Sous-Catégorie, Prix, Quantité, Total. Ajouter donc d'abord un **label et champ texte** au formulaire, pour la **quantité**.

b. **Ajouter la table** sous le panneau Caractéristiques Produit. Agrandir la fenêtre et ajouter le composant `Table` de la palette. On va définir notre modèle de table directement avec l'IDE : dans la propriété `Model`, choisir '`Table model customizer`' et saisir le nombre de colonnes (6), leur nom et leur type. Seuls le prix et la quantité seront modifiables directement dans la table.

c. Faire en sorte que l'utilisateur puisse sélectionner **plusieurs lignes à la fois**, par ex. pour les supprimer.

- Modifier l'événement associé au **bouton `Ajouter`**, qui va ajouter une ligne dans la `JTable` (supprimer le code du TP3 qui affichait les données).
  On va donc transférer les informations de la couche Vue (champs `textField` et `comboBox`) vers les données de la table, donc dans le Modèle.

  Utiliser pour cela la méthode `addRow()` du modèle, qu'il faut donc récupérer. Q1 : que prend `addRow()` en paramètre ?

  On enverra donc une instance de la classe attendue avec `new àVousDeTrouver[] {valeur_premiere_colonne, valeur_deuxieme_colonne, … }`. Pour les valeurs, on va récupérer les données saisies par l'utilisateur dans les champs de l'interface avec les méthodes `getText()` pour les `textField`, la méthode `getSelectedItem()` pour la `comboBox`.

  Attention, il faut envoyer les données *du type attendu* par les colonnes de la table : on transformera un élément de la `comboBox` en `String` avec `toString()`, et on utilisera les méthodes `parseXXX()` des wrappers (`Double`, `Integer`) pour extraire un réel/entier d'une chaîne.

- Il s'agit maintenant de **synchroniser les champs du formulaire avec la table,** quand on clique sur une ligne de la table. La démarche va être la suivante :
  – on créé un écouteur sur un clic souris, effectué sur la table ;
  – on lit les données de la ligne sélectionnée ;
  – on place ces données dans les bons champs du formulaire, pour que l'utilisateur puisse par exemple les modifier.

  - Définir un écouteur d'un clic sur la table : sélectionner la table, dans les `Events`, choisir **`MouseClicked`** de l'adaptateur `MouseAdapter`, et ajouter un 'handler' appelé par ex. `synchroniser()`.
  - Récupérer le modèle de la table, et lire les données avec les méthodes `getValueAt(ligne, col)`. L'argument 'ligne' sera défini par `getSelectedRow()` de la table. La 'colonne' sera successivement 0, 1 … 4 pour les valeurs des colonnes.
  - Affecter chaque `textField` avec ces valeurs en utilisant `toString`() quand nécessaire.

- Action du bouton **Supprimer**
  - Définir la méthode qui permet de **supprimer la ligne sélectionnée** par l'utilisateur quand ce dernier clique sur le bouton « `Supprimer` ». Afficher un *warning* si l'utilisateur clique sur le bouton supprimer sans sélectionner au moins une ligne de la table. Faire qu'on puisse supprimer plusieurs lignes à la fois.

- On souhaite vider les champs et placer le curseur dans le champ *libellé* après avoir supprimé la saisie courante. Pour cela, utilisez la **gestion du focus**, qui marche en 2 temps : avant d'affecter le focus à un `textField` avec `requestFocus`(), il faut vérifier que le champ est 'focusable' avec `isFocusable`(). S'il ne l'est pas, on le rend : `setFocusable (true).`

- Action du bouton **Modifier**
  - Ajouter une bulle d'aide au bouton **`Modifier`** qui explique à l'utilisateur qu'il doit modifier les valeurs dans le formulaire, puis cliquer sur *Modifier* pour changer les valeurs dans la table (utiliser la propriété du composant bouton : `toolTipText`).
  - Définir en conséquence l'action du bouton ***Modifier***, qui va modifier les valeurs de la ligne sélectionnée avec les valeurs du formulaire, quand l'utilisateur clique sur le bouton *Modifier*. C'est un code très similaire à celui du bouton *Ajouter*.

    Testez votre interface avec le bouton *Modifier*. Q2 : quel type d'exception est levée quand on n'a sélectionné aucune ligne de la table ? Ajoutez un message d'erreur si l'utilisateur clique sur le bouton modifier sans sélectionner une ligne.

FACULTATIF… Barre d'outils et Export Excel

- Tester si la ligne **existe déjà** dans la table avant de l'ajouter
  - (même libellé, même catégorie, même sous-catégorie)

- Ajouter les boutons 'Test Son1' et 'Test Son2' dans une **barre d'outils** plutôt que par menu, quand on choisit un casque audio.

  a- Créer la barre d'outils avec un bouton (sans bord), et ajoutez-la à votre fenêtre.

  b- Modifier votre code pour permettre le test des sons **en boucle** continue (possible avec la fonction `clip.loop()`), ou un message d'erreur « Pas de fichier son » si jamais il manque. Gérer le texte du bouton pour qu'il passe à 'Arrêter' le son, et implémenter la gestion qui convient.

- Ajouter **l'export Excel** du contenu de la table.

Quelques indices :

- (Nul besoin de passer par l'API `jexcelapi` !)
- Une solution possible est, dans l'écouteur approprié, de choisir le fichier pour l'export, puis d'exporter le contenu de la table dans le fichier comme illustré ici :

```
void xxxxActionPerformed(ActionEvent e) {
File fichier = this.choisirFichier() ;      // méthode à écrire
exporterExcel( maTableProduits, fichier) ; // méthode à écrire
}
```

- Pour `choisirFichier()`, on utilisera une `FileDialog` : étudier la javadoc.

- Dans la méthode `exporterExcel()`, il suffit décrire les entêtes de colonnes d'abord, séparées par des tabulations dans par ex. un fichier texte type FileWriter :

```
FileWriter fo = new FileWriter(fichier); // fo pour 'fichier out'

 for (int i = 0; i < monModeleTable.getColumnCount(); i++) {
        fo.write(monModeleTable.getColumnName(i) + "\t");
 }
```

- Puis d'y placer un caractère de *fin de ligne* :

```
fo.write("\n");
```

- Ensuite on écrit chaque ligne de la table, avec une tabulation entre les champs :

```
fo.write(monModeleTable.getValueAt(i, j).toString() + "\t");
```

- Puis de placer une *fin de ligne* entre chaque ligne de la table :

```
fo.write("\n");
```

- On lance le tableur avec l'instruction :

```
Desktop.getDesktop().open(fichier); // le File fichier précédent
```

NOTA : l'OS reconnaît qu'il s'agit du tableur par défaut, par l'extension du fichier (.xls).

Q3 – Que se passe-t-il si on ne mentionne pas d'extension ?