DataBinding

1 Apprentissage

L'objectif de cette séance est de travailler avec la liaison de données (*databinding*).

1.1 GIT

Reprenez votre dépôt git et créez une branche "tp6" depuis la branche `master`.
Copiez-y le projet que vous trouverez sur eCampus.

1.2 Étude des fichiers présents

Regardez attentivement le *package* `modele` et ses fichiers `Produit.java` et `Ligne.java`. La **Ligne** possède plusieurs propriétés inutilisées. Une classe de Test JUnit est fournie. Utilisez votre IDE pour l'exécuter (échec).

1.3 Totaux et observateurs...

L'objectif de cet exercice est de faire calculer les totaux et sous-totaux automatiquement : le total HT de chaque ligne et leur total TTC. On pourrait pour cela reposer sur les *setters*, mais cela ne protège pas d'une modification "sauvage" du prix d'un produit sans passer par la facture !

Heureusement, Java-FX vous propose en standard le patron *Observateur* (voir cour de S3). Vous allez donc utiliser des propriétés Java-Fx pour automatiser le calcul, et en définitive, la mise à jour de l'interface graphique :

— Grâce à la classe **Bindings** et à sa méthode statique `multiply`, créez une **NumberExpression** correspondant au calcul $Qté * Prix$; vous avez calculé le prix HT de la ligne. En testant votre classe, vous verrez que ce calcul est mis à jour automatiquement dès que vous modifiez la quantité ou le prix. Vous n'avez donc à l'écrire qu'une seule fois !
— De façon similaire, définissez le prix TTC de la ligne avec $prixHT * TVA$.

Normalement, la classe de Tests devrait passer sans souci, sauf pour le test `mathTestSetProduit`.

Pour ce dernier test, il est nécessaire que les propriétés "suivent" le produit. Il faut pour cela utiliser une autre méthode statique de la classe **Bindings** : `select`.
Cette méthode prend un objet observable, et traverse une suite de propriétés données par leur nom. Par exemple, `Bindings.selectFloat(ligne.produitProperty(), "prix")` retournera un **FloatBinding** qui dépendra de la ligne, mais aussi du produit qu'elle contient.

N'oubliez pas de *commiter* vos fichiers !

1.4 Intégration dans l'IHM

Dans le projet, vous avez également une application Java-FX sommaire permettant de créer une "facture".
La vue `facture.fxml` contient une **VBox**, une **HBox** avec un **Label**, un **TextField** en lecture seule, un **Button** et surtout une **TableView**. Elle utilise aussi une feuille de style, mais on y reviendra plus tard...

Le composant **TableView** est un peu complexe ; il regroupe des **TableColumn** qui présenteront des lignes de données à l'utilisateur, en lui permettant de les trier ou de les déplacer à sa guise.
Vous noterez que la table est définie comme une **TableView<Ligne>** dans le contrôleur FXML, ce qui indique que la table affichera des éléments de la classe **Ligne**.
De la même façon, chaque colonne est définie en indiquant le type de l'objet de chaque ligne de la table, mais aussi le type spécifique de la valeur qui sera affichée dans la colonne.

Si vous exécutez l'application, vous verrez une table vide, mais fonctionnelle. Vous pouvez déjà réordonner, trier ou redimensionner les colonnes !

1.4.1 remplissage de la table

Afin que cette table ne reste pas vide, vous allez compléter la méthode `onAjouter` du contrôleur :

— Créez une nouvelle instance de **Ligne**.
Vous pouvez utiliser un produit de la fabrique **FabriqueProduits** ou en créer un vous-mêmes.
Tirez une quantité aléatoire grâce à la méthode `nextInt(max)` de la classe **Random**.

— Ajoutez aux `items` de la table votre ligne.

Testez votre application. Vous noterez que la table n'est plus vide, mais qu'elle n'affiche rien… c'est normal.
Vous devez donc observer des lignes de couleurs alternées. Celle-ci répond en effet à une feuille de style (*stylesheet*).
Votre projet modifie la feuille par défaut avec le fichier `facture.css`. Vous y verrez en particulier la pseudo-classe `hover` définie pour la classe `table-cell`. Passez votre souris sur la table pour la voir en action…

1.4.2 affichage des colonnes

Afin que les colonnes ne soient pas vides, il faut leur expliquer quoi afficher avec la méthode `setCellValueFactory` de chaque **TableColumn** : Cette *fabrique* (voir R3.04) sera appelée pour chaque ligne de chaque colonne afin de lier la vue aux propriétés du modèle.

— La quantité est simplement définie comme une propriété de la classe **Ligne**. Vous allez donc pouvoir utiliser "`qte.setCellValueFactory(new PropertyValueFactory<>("qte"));`"
Cette méthode va aller chercher la propriété `qte` de la classe **Ligne** grâce à ses *getters* par réflexion, pendant l'exécution. C'est donc pratique, mais peu efficace et non *typesafe*.
Essayez de faire une faute dans le nom de la propriété : ça compile, mais ça génère une exception à l'exécution.

— Le produit est similaire, mais vous allez le faire autrement !
Créez donc une **Callback<TableColumn.CellDataFeatures<Ligne, Produit>, ObservableValue<Produit> >** et qui servira de fabrique à valeur : Elle reçoit un descripteur de contenu de cellule et fournit un **Produit** observable.
Le descripteur possède une méthode `getValue` qui retourne la **Ligne** courante.
Utilisez cette **Ligne** pour retourner la propriété `produit` de la **Ligne**.
Cette méthode est moins pratique, mais elle est *typesafe*. Il est impossible de se tromper dans le nom de la propriété, par exemple. Elle est également plus efficace à l'exécution et ne requiert pas l'ouverture du *package* à la réflection.

— Le prix unitaire est une propriété du **Produit**, qui est dans une propriété de la **Ligne**.
Utilisez la technique précédente pour la retourner. Vous pouvez aussi utiliser une lambda pour simplifier le code.

— Définissez enfin les colonnes des prix HT et TTC avec la technique de votre choix.

Testez l'application. L'affichage du produit laisse à désirer, car Java-FX utilise le `toString` pour afficher les objets qu'il ne sait pas afficher nativement. On laissera ce problème de côté pour le moment.

Pensez à *commiter* vos fichiers.

1.4.3 les colonnes modifiables

Vous avez probablement vu dans le `FXML` que les deux premières colonnes sont `editable`. Pourtant, cela ne fonctionne pas. Java-FX a en effet besoin de savoir comment vous voulez modifier ces cellules.

Vous allez pour cela définir une nouvelle fabrique, qui définit non seulement comment les cellules sont modifiables mais aussi comment elles doivent s'afficher :

— La quantité est un simple entier ; on va donc utiliser un **TextField** pour le modifier.
Java-Fx possède une classe à cet effet : **TextFieldTableCell**. Cependant, la conversion entre un **Integer** et un **String** n'est pas directe ; on va donc devoir lui fournir en plus un **IntegerStringConverter**.
Utilisez donc l'une des deux formulations suivantes :
`qte.setCellFactory(TextFieldTableCell.forTableColumn(new IntegerStringConverter()));` ou
`qte.setCellFactory(cell -> new TextFieldTableCell<>(new IntegerStringConverter()));`

— Pour modifier le produit, on va proposer une liste de choix à l'utilisateur.
Java-Fx possède une classe à cet effet : **ChoiceBoxTableCell**. Cependant, la conversion entre un **Produit** et un **String** n'est pas directe ; on va donc devoir lui fournir en plus un **StringConverter<Produit>**. Écrivez dans cette classe les deux méthodes `toString`, qui retournera simplement le nom du **Produit**, et `fromString`, qui recherchera dans la liste le **Produit** qui a le nom donné.
ChoiceBoxTableCell n'utilise malheureusement pas une **List** d'éléments, mais une **ObservableList**. Heureusement, la classe **FXCollections** contient une méthode `observableList` qui fait la conversion depuis la liste de produits (voir **FabriqueProduits**.`getProduits()`) pour vous.

Testez et *commitez* votre projet.

1.4.4 la somme globale

Modifiez la fonction `onAjoute` de façon à ce que le **TextField** de l'IHM affiche bien la somme de la facture toujours juste. (voir **Bindings**.`add`)

Vérifiez que la modification d'une quantité ou d'un produit d'une ligne modifie bien le total !

1.4.5 les colonnes monétaires

L'affichage des colonnes "*float*" laisse à désirer. On aimerait un affichage à deux décimales, aligné à droite, et en rouge pour les nombres négatifs.
Il n'existe évidemment pas de classe toute faite qui fasse cela. La méthode `setCellFactory` vous permet cependant d'utiliser toute classe qui hérite de **TableCell<Ligne, C>**.

Vous allez donc créer une classe qui hérite de **TableCell<T, Number>**, et qui va redéfinir la fonction `updateItem`. Cette classe sert de base à toutes les cellules d'une table, et hérite elle-même de **Labelled**. C'est donc un **Label** remplaçable par autre chose à volonté.

Dans la méthode `updateItem`, vous allez donc :

— Appeler la méthode héritée. (super)
— Aligner le texte à droite (`setAlignment`).
(Peut aussi être fait une fois pour toute dans le constructeur…)
— Tester si la cellule est vide (`empty`) ou si l'objet contenu est `null`.
  — Si oui, faire `setGraphic(null);setText(null);` (c'est important pour éviter les *glitches*)
  — Si non, formater le texte de la cellule (`String.format`, **NumberFormat**, …)
— Colorer le texte si la valeur est négative :
Pour la colorisation du texte, on peut utiliser `setTextFill`, ou utiliser la feuille de style.
Pour cette dernière solution, on peut utiliser la méthode `pseudoClassStateChanged` avec en paramètre la pseudo-classe CSS (`PseudoClass negatifCSS = PseudoClass.getPseudoClass("negatif")`) et un booléen qui l'active ou non.

Testez votre application, *commitez* et poussez votre travail !

2 Réutilisation

Revenez à la branche “gribouille_stable” et créez une nouvelle branche “gribouille_tp6”.

2.1 Modèle

Ajoutez le code que vous trouverez sur eCampus aux différentes classes du modèle. Mettez à jour les importations et ajoutez les propriétés manquantes.

2.2 Sauvegarde du fichier

Lorsque le menu “Sauvegarder” est activé, ouvrez un **FileChooser** en mode sauvegarde. Vous pourrez ensuite appeler la méthode `sauveSous` de la classe **Dessin** en lui passant le résultat de la méthode `getAbsolutePath()` du fichier choisi. Assurez-vous bien que le titre de la fenêtre change…

Il est certainement préférable de créer une méthode `sauvegarde()` dans le **Controleur**…

2.3 Titre de fenêtre intelligent

La classe **Dessin** contient une propriété `estModifie`.
Modifiez le titre de la fenêtre pour ajouter une étoile ‘*’ au nom du dessin s’il est modifié.
Vous pourrez utiliser **Bindings**.`when` qui permet de faire un si-alors-sinon dans une propriété.
Bindings.`concat` permet de coller deux chaînes de caractères, observables ou non.

2.4 Chargement du fichier

Lorsque le menu “Charger” est activé, ouvrez un nouveau **FileChooser**, mais en mode chargement, cette fois. Vous pourrez ensuite appeler la méthode `charge` de la classe **Dessin** en lui passant le résultat de la méthode `getAbsolutePath()` du fichier choisi. N’oubliez pas d’effacer le **Canvas** et de demander à le redessiner.

Il est certainement préférable de créer une méthode `charge()` dans le **Controleur**…

2.5 Quitter

Vous pouvez maintenant finaliser la méthode `onQuitter` du **Controleur** :
Si le dessin n’a pas été modifié, retourner simplement `true`.
Sinon, proposez une boîte de dialogue à l’utilisateur permettant de :

1. Quitter sans sauvegarder (retourner `true`)
2. Quitter en sauvegardant (appeler la méthode `sauvegarde` et retourner `true`)
   Attention à ne pas retourner `true` en cas d’exception pendant la sauvegarde!
3. Ne pas quitter (retourner `false`)

3 Rendu

N’oubliez pas de *commiter* vos fichiers régulièrement, et de faire un *merge* et des *push* à la fin!

4 Ce qui manque encore

Si on avait une séance de plus, il faudrait ajouter à Gribouille :

- La gestion du **ColorPicker** permettant à l'utilisateur de choisir une couleur.
- Un menu permettant d'effacer le dessin.
- L'exportation du dessin sous forme d'une image :
  - Ouvrir un dialogue **FileChooser**
  - Personnaliser son filtre avec un **FileChooser.ExtensionFilter**
  - Créer un *snapshot* du **Canvas** :
    ```
    WritableImage image = dessin.snapshot(new SnapshotParameters(), null);
    ```
  - Exporter l'image (on doit hélas passer par Swing pour cela) :
    ```
    javax.imageio.ImageIO.write(SwingFXUtils.fromFXImage(image,null), format, file);
    ```
  - Attraper l'exception et afficher une **Alert**.

Trois autres éléments n'ont pas été couverts, mais n'ont pas leur place dans Gribouille :

- Les animations (voir PDF annexe)
- La 3D (voir PDF annexe)
- Les graphiques / courbes (pas encore de PDF)