Conception et Programmation Objet Avancées

Compléments pour le projet

Créer des boîtes de dialogues (1)

- Une boîte de dialogue est une fenêtre modale (tant qu'elle n'est pas fermée, pas d'interaction sur la fenêtre mère)
- La boîte de dialogue hérite de *Stage*
- Le chargement du fichier FXML se fait comme pour l'application (fenêtre principale)

```
public class VueProduits extends Stage {

  public VueProduits() {
    try {
      final URL fxmlURL=
                    getClass().getResource("/vue/produits.fxml");
      final FXMLLoader fxmlLoader = new FXMLLoader(fxmlURL);
      final VBox node = (VBox)fxmlLoader.load();
      Scene scene = new Scene(node);
```

Créer des boîtes de dialogues (2)

- Gérer la modalité

```
this.initModality(Modality.APPLICATION_MODAL);
```

- Fermer la fenêtre et revenir à la fenêtre mère (ça se passe dans le contrôleur)

```
/**
* Réaction au bouton Retour
*/
public void fermeDialogue() {this.vue.close();}
```

- (méthode déclarée dans la vue FXML, onAction d'un bouton « Retour »)

Créer des boîtes de dialogues (3)

- Donc… Le contrôleur doit connaître la vue

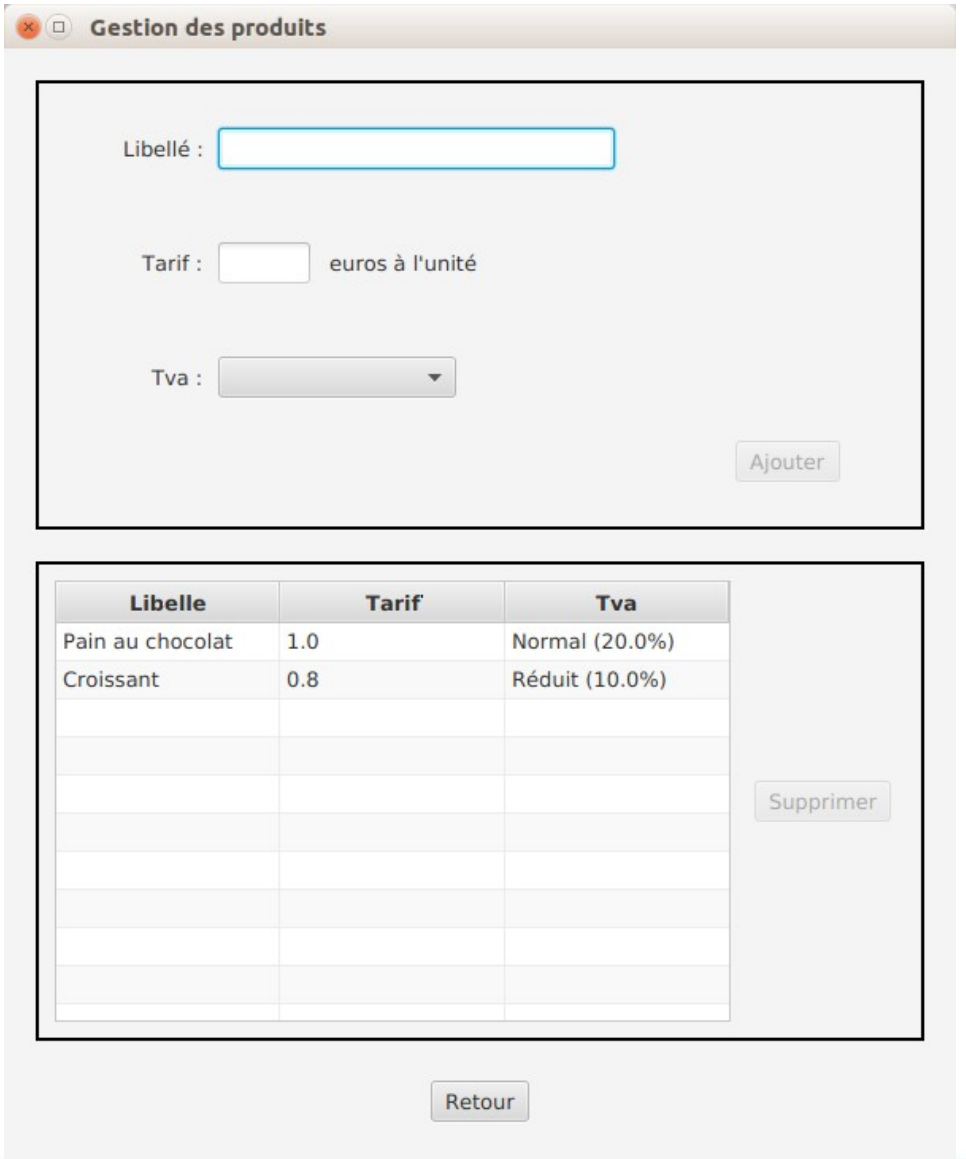
```
public class VueProduits extends Stage {

  public VueProduits() {

…

    CtrlProduits controleur = fxmlLoader.getController();
    controleur.setVue(this);

…

    this.show();
  }
```

Affichage d'alertes

```
String erreur = "...";
Alert alert=new Alert(Alert.AlertType.ERROR);
alert.initOwner(this.vue);
alert.setTitle("Erreur lors de la saisie");
alert.setHeaderText("Un ou plusieurs champs
sont mal remplis.");
alert.setContentText(erreur);
alert.showAndWait();
```

Les tables de données (1)

- Ce que l'on veut obtenir



Les tables de données (2)

- FXML : La table est une *TableView*
- Code dans le contrôleur

```
@FXML
private TableView<Produit> tblProduits;
```

Les tables de données (3)

- Création des colonnes :
  - La colonne affiche une chaîne appartenant à une instance de produit, l'entête est « Libellé »

```
TableColumn<Produit, String> colLibelle =
                        new TableColumn<>("Libellé");
```

  - La cellule va utiliser l'attribut *libelle* de Produit

```
colLibelle.setCellValueFactory(
 new PropertyValueFactory<Produit, String>("libelle"));
```

  - Attention : ne fonctionne que si le setter de l'attribut *libelle* s'appelle *getLibelle()*
  - ~~*get_libelle()*~~, ~~*getlibelle()*~~, *etc.*

Les tables de données (4)

- Ajout des colonnes à la table

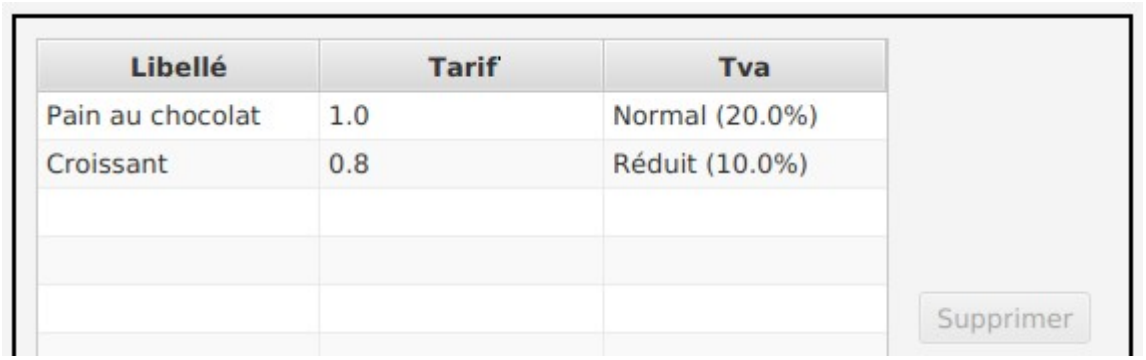
```
this.tblProduits.getColumns().setAll(
                    colLibelle, colTarif, colTva);
```

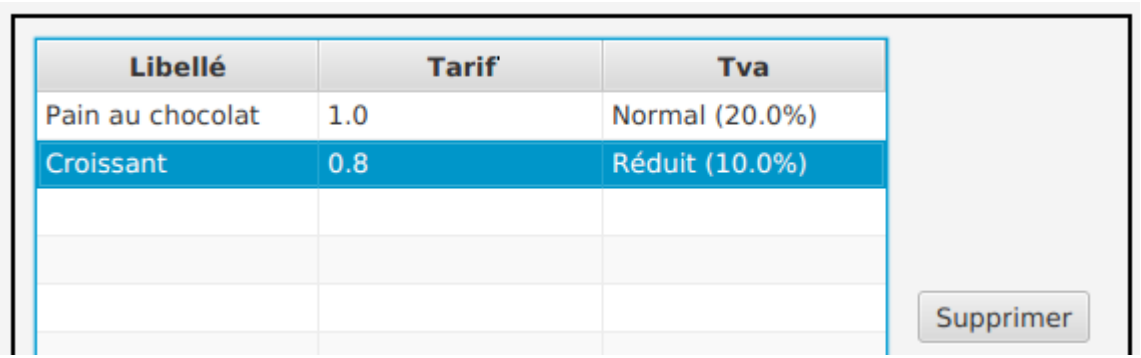
- Ajout des données

```
this.tblProduits.getItems().addAll(
    DAOFactory.getDAOFactory().getProduitDAO().findAll());
```

Les tables de données (5)

- Ce que l'on veut faire :





- => Ajout d'un *Listener* sur la sélection

Les tables de données (6)

- Ecouter les sélections (~~SceneBuilder~~)

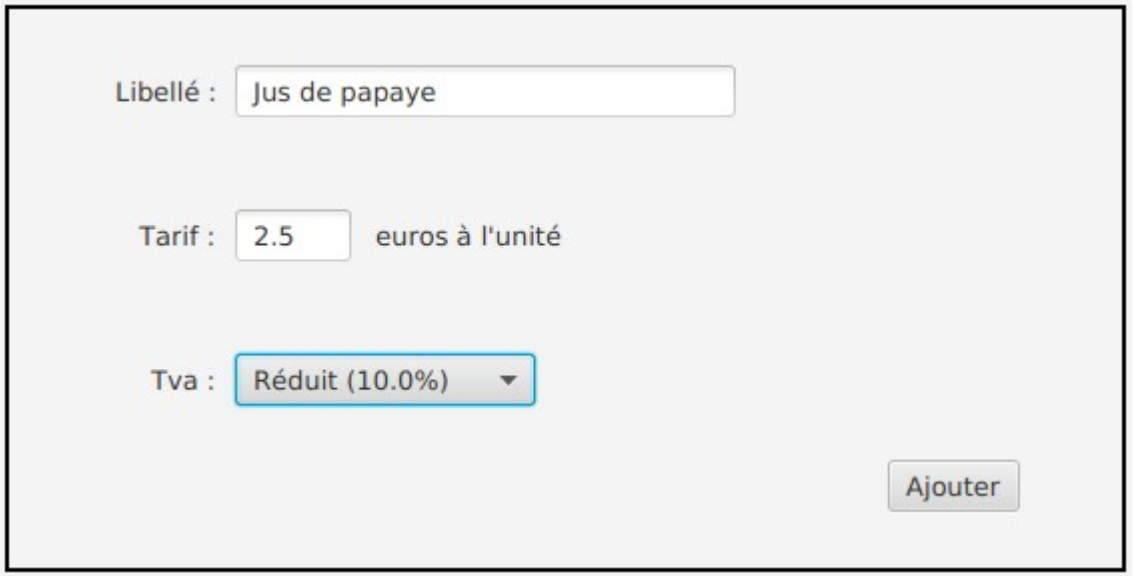
```
this.tblProduits.getSelectionModel().
                selectedItemProperty().addListener(this);
```

- Implémenter le listener

```
public class CtrlProduits implements ChangeListener<Produit>
{
    …
    public void changed(
          ObservableValue<? extends Produit> observable,
          Produit oldValue, Produit newValue) {

       this.btnSupprimer.setDisable(newValue == null);
```

Ecouter la saisie (1)

- Activation du bouton uniquement si...



- … les champs obligatoires sont remplis

Ecouter la saisie (2)

- Ajout du listener (~~SceneBuilder~~)

```
this.txtLibelle.textProperty().addListener(this);
this.txtTarif.textProperty().addListener(this);
this.cbxTva.getSelectionModel().selectedItemProperty()
.addListener(this) ;
```

- Implémenter deux fois le listener (String, Tva)

```
public class CtrlProduits implements
        ChangeListener<Produit>,
        ChangeListener<String>,
        ChangeListener<Tva> {
```

- (on en a déjà un pour la sélection de table)
- => Conflit : la classe ne peut implémenter trois versions de ChangeListener

Ecouter la saisie (3)

- Solution ?
- Créer une classe à part juste pour implémenter chaque *Listener*
- Ou : utiliser les *lambdas expression* (Java 8) pour deux des *Listeners*
- *1 objet s'adresse à ChangeListener<Produit>*
- *1 autre à ChangeListener<Tva>*
- *2 objets s'adressent à ChangeListener<String>*
- => Les listeners de table et de liste déroulante sont gérés sous forme de *lambda expression*

=> Ecouter la sélection de table

- Sans lambda expression

```
public class CtrlProduits implements ChangeListener<Produit>
{
    this.tblProduits.getSelectionModel().
                    selectedItemProperty().addListener(this);

    public void changed(
          ObservableValue<? extends Produit> observable,
          Produit oldValue, Produit newValue) {
       this.btnSupprimer.setDisable(newValue == null);
```

- Avec lambda expression

```
this.tblProduits.getSelectionModel().selectedItemProperty
().addListener(
    (observable, oldValue, newValue) -> {
         this.btnSupprimer.setDisable(newValue == null);
     });
```

Connexion toujours ouverte (1)

- Rappel : singleton

```
public class Connexion {

  private static Connexion instance;

  public static Connexion getInstance() {
    if (instance == null) {
      instance = new Connexion();
    }
    return instance;
  }

  private String url, login, pwd;
  private Connection maConnexion;

  private Connexion() {
    this.litFichier();
  }
```

Connexion toujours ouverte (2)

- Création de la connexion SI NECESSAIRE

```
public Connection creeConnexion() {

  try {
    if (maConnexion == null || maConnexion.isClosed()) {
      maConnexion = DriverManager.getConnection(this.url,
                                        this.login, this.pwd);
    }
  } catch (SQLException sqle) {
    System.out.println("Erreur connexion" +
                                      sqle.getMessage());
  }

  return maConnexion;
}
```

- => *close()* n'est plus nécessaire dans les DAO