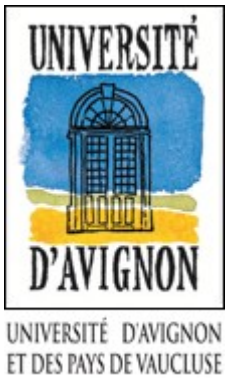
Prototypage et Interfaces utilisateurs



Master Informatique M1 2016-2017 - Université d'Avignon

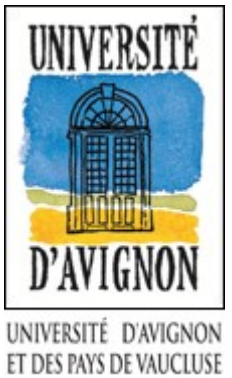
GL Avancé: Prototypage et interfaces utilisateur

Objectifs du cours

- **Prototypage simple (dit horizontal):**
  - Conception d'une maquette statique (mockup)
  - Traduction des besoins utilisateur en composants graphiques
- **Prototypage fonctionnel (dit vertical):**
  - Ajout d'interactivité : maquette dynamique
  - Binding du modèle de données avec l'interface
  - Réalisation d'un scénario d'utilisation (use case)
- **Prototypage d'échelle sur une plateforme dédiée:**
  - Utilisation d'un environnement de fenêtrage de haut niveau
  - Découpage en plugins
  - Branding et personnalisation
  - Déploiement et livraison du prototype

2ème partie:

Prototypage fonctionnel



Prototypage fonctionnel

- **Prototype vertical:**

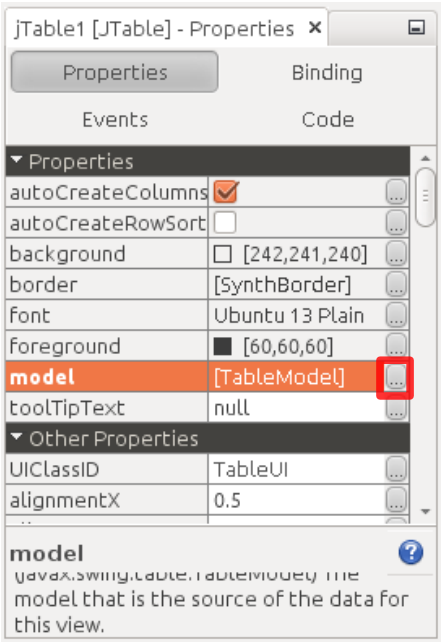
  - Maquette dynamique et interactive
  - Connexion des composants à des données concrètes
  - Simulation de certaines des fonctionnalités
  - Déroulement de scénarii d'utilisation
  - Vérifier le rendu final et l'utilisabilité de l'application
  - Estimation du niveau de performance et de réactivité

→ Apporte une valeur ajouté au prototype avec un minimum de codage

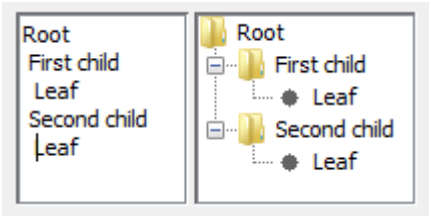
GL Avancé: Prototypage et interfaces utilisateur

Les modèles de données

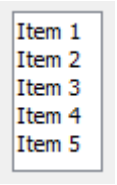
- Pré-remplir les modèles de données
  - ✓ Valider le choix du composant graphique le mieux adapté au besoin
  - ✓ Simuler un chargement pour anticiper la bonne lisibilité
  - ✓ Éditeur de modèle intégré à NetBeans



- Éditeur de TreeModel :
  - ✓ Arborescence décrite par indentation
  - ✓ Une racine unique



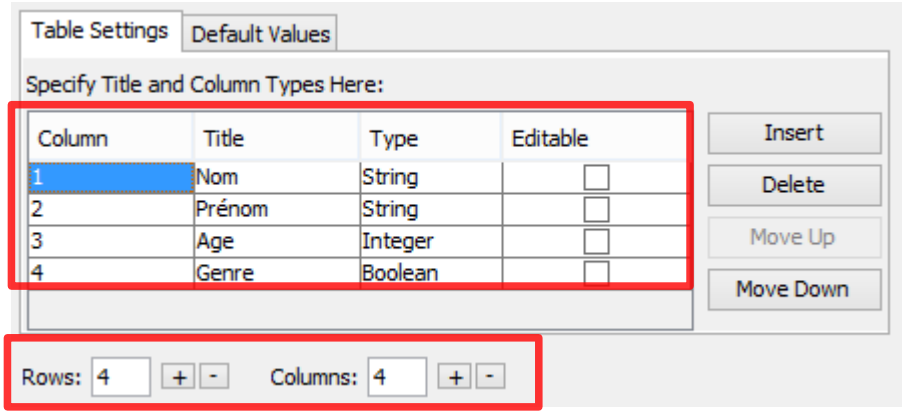
- Éditeur de ListModel et ComboBoxModel:
  - ✓ Énumération des éléments

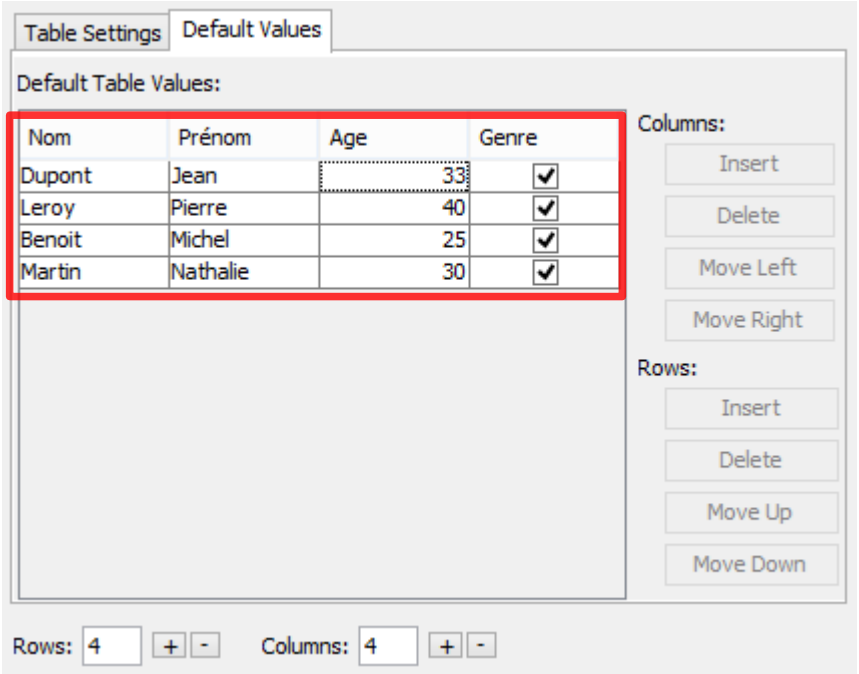


GL Avancé: Prototypage et interfaces utilisateur

Les modèles de données

- Éditeur de TreeModel :
  - ✓ Nombre de lignes et de colonnes
  - ✓ Nom des colonnes
  - ✓ Type des colonnes
  - ✓ Rendre éditable les cellules
  - ✓ Insertion, suppression, déplacement des colonnes
  - ✓ Ajout de valeurs par défaut
  - ✓ Intégration du modèle dans la zone de design

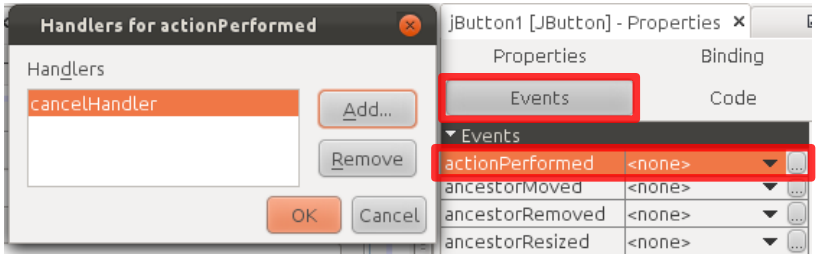




GL Avancé: Prototypage et interfaces utilisateur

Interactivité avec NetBeans

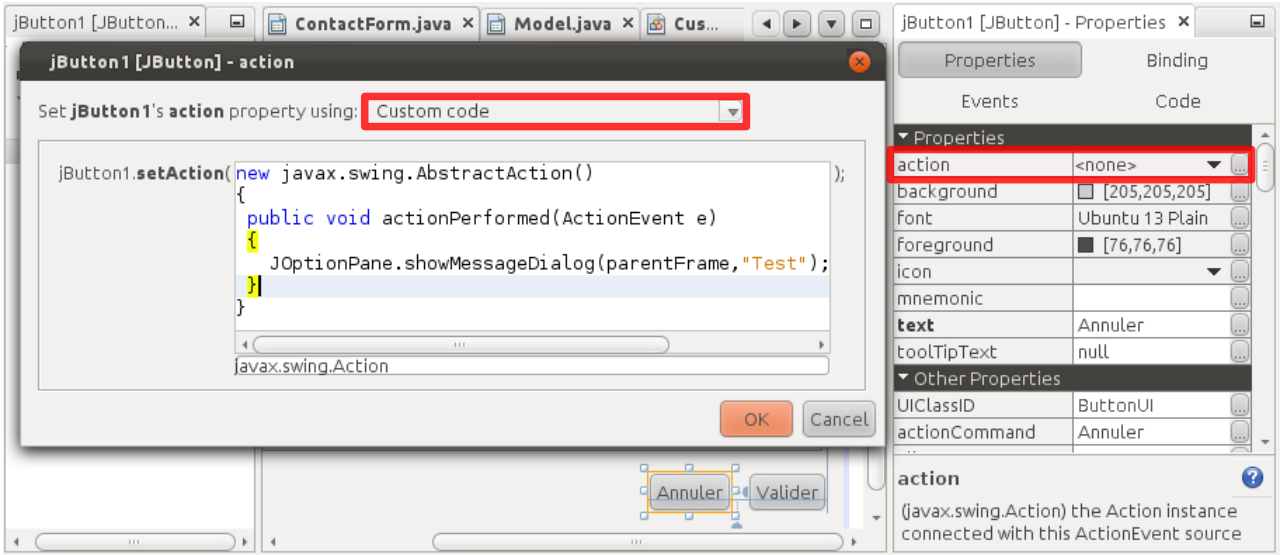
- Les handlers: méthode liée à l'écoute d'événements




```
private void initComponents() {
  ...
  jButton1.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(java.awt.event.ActionEvent evt) {
          cancelHandler(evt);
      }
  });
}
private void cancelHandler(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
}
```

Code généré

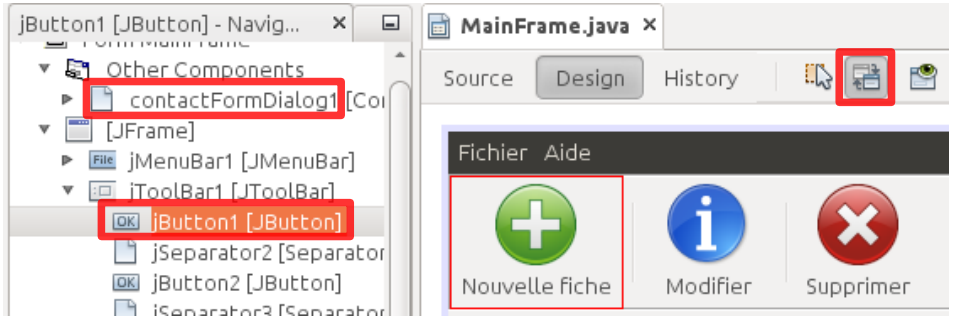
- Les actions: classe indépendante dérivée de l'interface « javax.swing.Action »

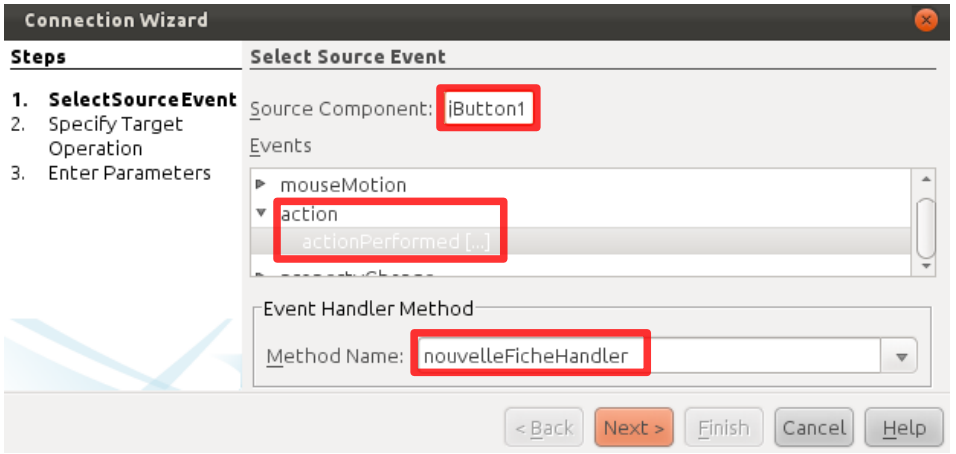


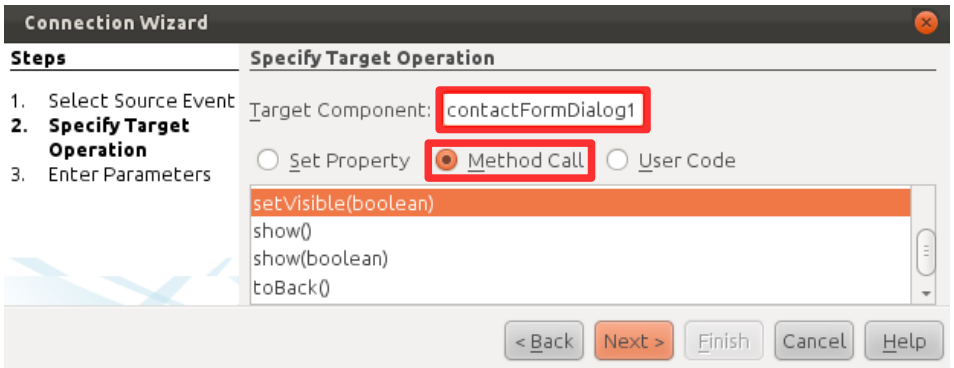
GL Avancé: Prototypage et interfaces utilisateur

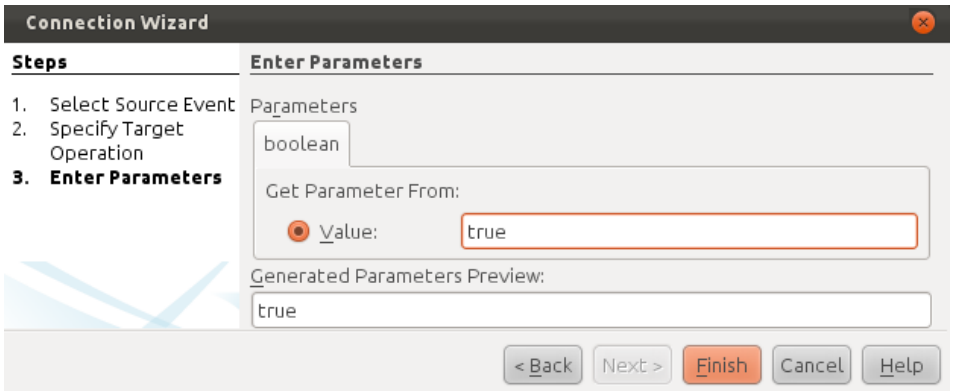
Interactivité assistée avec Matysse

- **Éditeur en mode connexion :**
  - lier un handler d'un élément avec une propriété ou un appel de méthode d'un autre élément
  - Exemple : déclencher l'ouverture d'une boîte de dialogue depuis un bouton





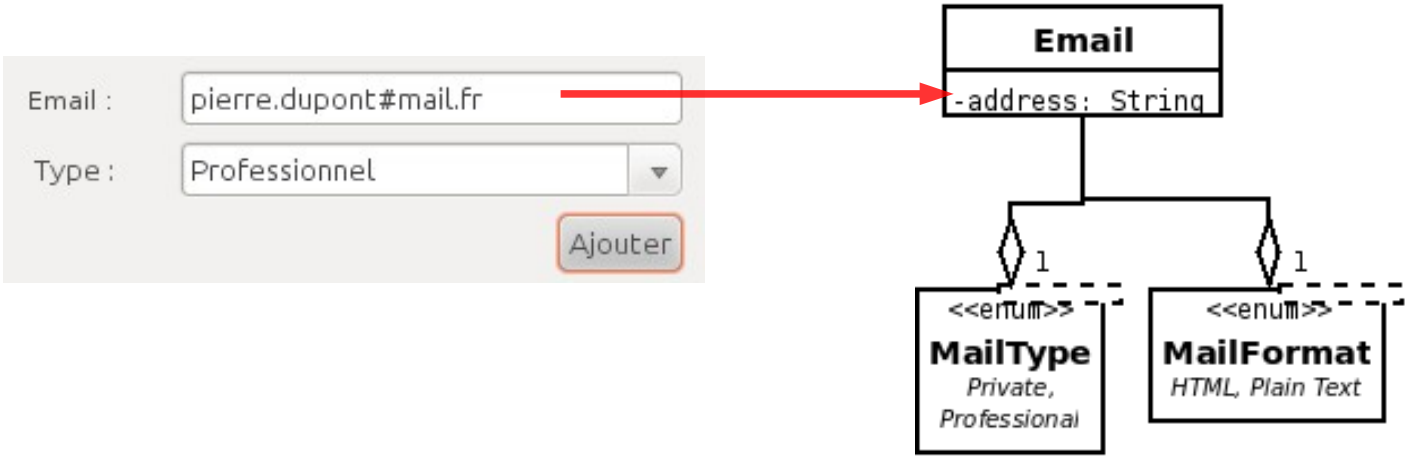




GL Avancé: Prototypage et interfaces utilisateur

Data Binding

- But: synchroniser automatiquement des données saisies



- Binding vers des propriétés d'objets ou des tables de base de données (Entity classes)
- Validation de la saisie avec notification d'erreur
- Conversion de la saisie vers le type de la donnée

Data Binding

- **Différents modes de synchronisation:**
  - Lecture/Écriture:
    - ✓ Synchronisation bi-directionnelle
    - ✓ Toute modification de l'un est immédiatement répercutée sur l'autre
  - Lecture seulement:
    - ✓ Lecture depuis la source
    - ✓ Seules les modifications de la propriété du bean sont appliquées au composant
  - Lecture unique:
    - ✓ Lecture depuis la source à l'initialisation du composant
    - ✓ Le composant charge la valeur stockée dans le bean une seule fois

Data Binding

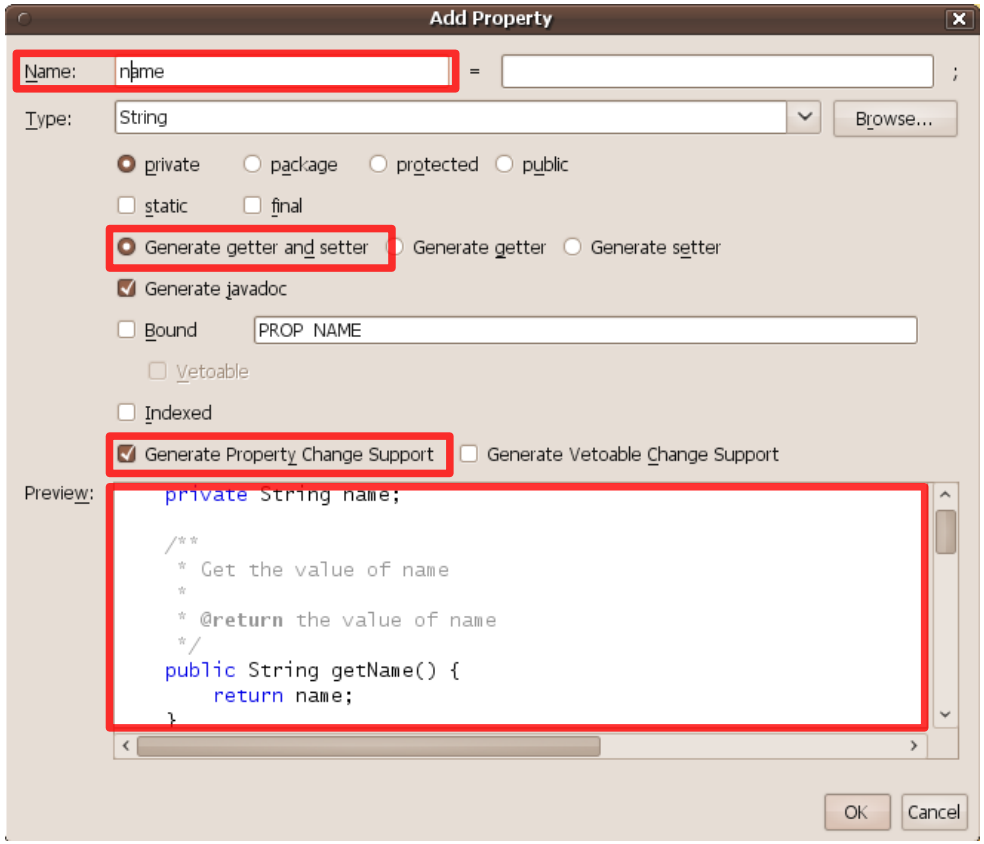
- **JavaBeans et propriétés**
  - Spécifications d'un « Bean » au sens Java **[Java Specification Request 295]**
    - ✓ Classe Java d'encapsulation des données
    - ✓ Des propriétés avec méthodes d'accès et d'écriture (Getter/Setter)
    - ✓ Un constructeur vide par défaut

```
public class Contact {
    private String name ;

    public Contact() {
        this("John");
    }

    public Contact(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

GL Avancé: Prototypage et interfaces utilisateur

Data Binding

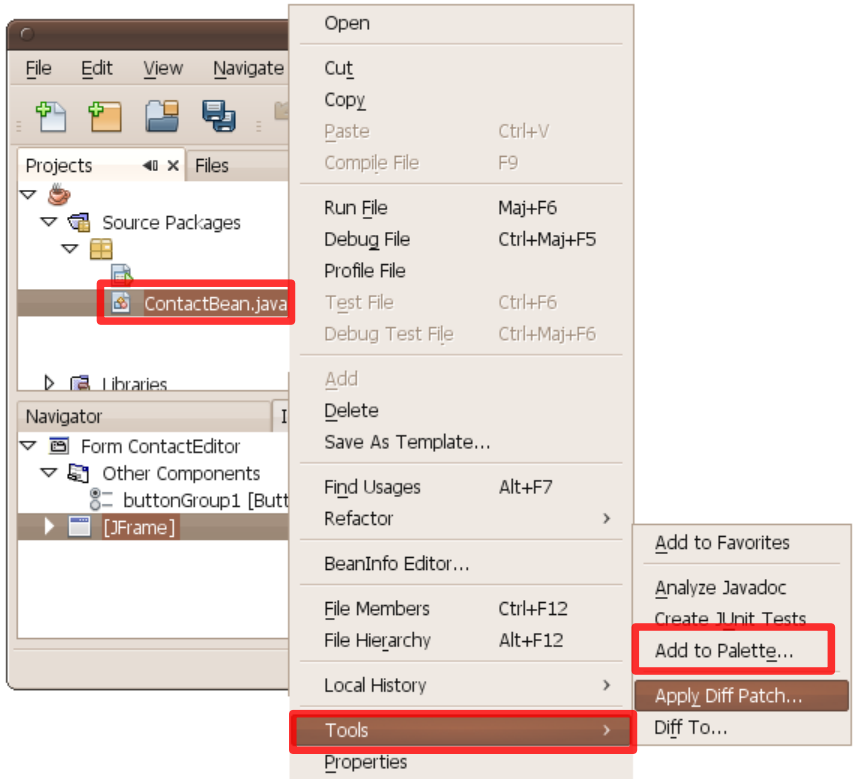
- **JavaBeans avec NetBeans**
  - Génération automatique des getters/setters
  - Génération automatique du PropertyChangeSupport pour notifications des modifications



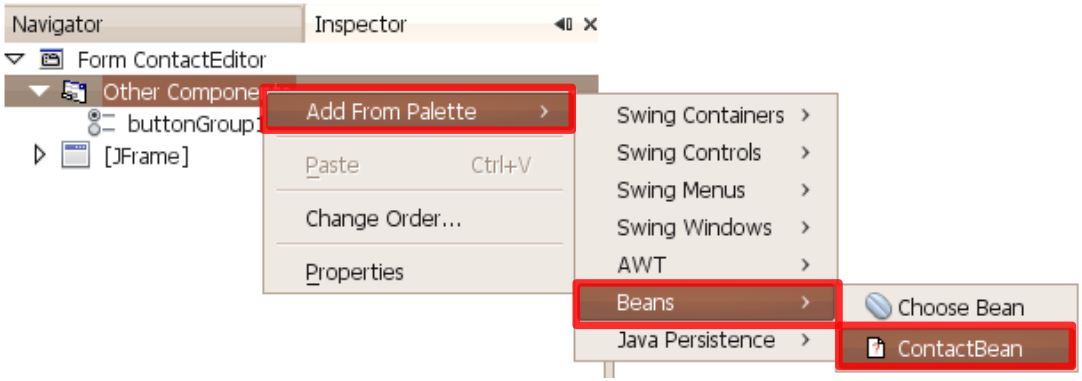
GL Avancé: Prototypage et interfaces utilisateur

Data Binding avec NetBeans

- Ajout du « bean » à la palette



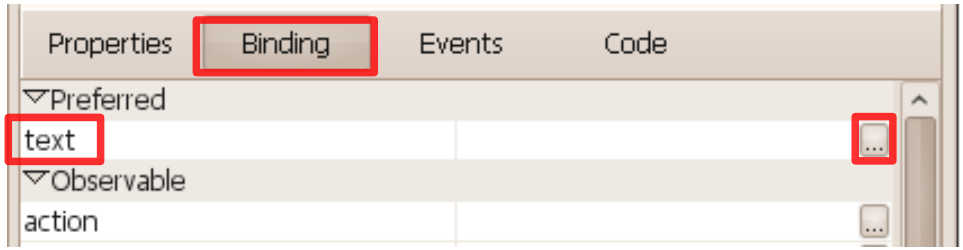
- Utilisation du « bean » depuis la palette



GL Avancé: Prototypage et interfaces utilisateur

Data Binding avec NetBeans

- **Configuration du Data Binding entre un composant et un bean**
  - Édition des propriétés de binding du composant Swing depuis la vue « Properties »



  - Binding entre la propriété « name » du bean et la propriété « text » du composant

GL Avancé: Prototypage et interfaces utilisateur

Data Binding avec NetBeans

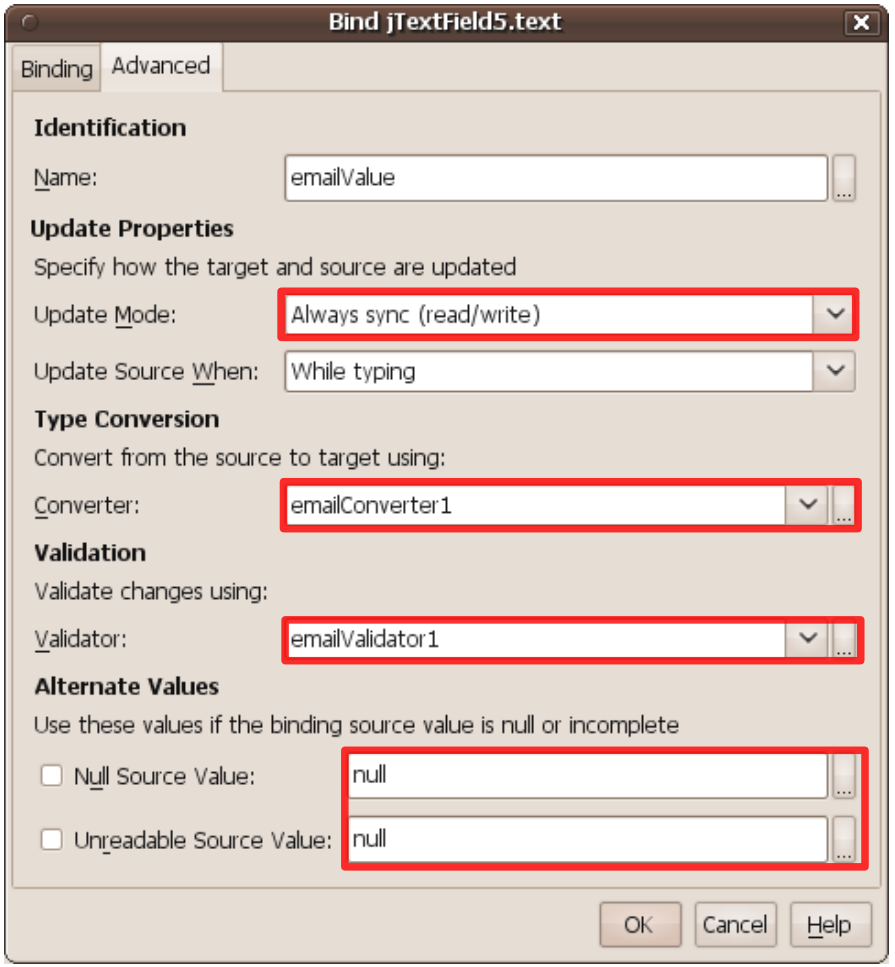
- Code java correspondant au binding généré par NetBeans

```
private void initComponents()
{
    bindingGroup = new org.jdesktop.beansbinding.BindingGroup();

    //Bind property « name » of source with property « text » of textfield target
    Binding binding = Bindings.createAutoBinding(
        UpdateStrategy.READ_WRITE,      // The update strategy for the binding
        contactBean,                    // The source object
        ELProperty.create("${name}"),   // The source property
        textField,                      // The target object
        BeanProperty.create("text"));   // The target property

    bindingGroup.addBinding(binding);

    bindingGroup.bind();
}
```

GL Avancé: Prototypage et interfaces utilisateur

Data Binding avec NetBeans

- Propriétés avancées:
  - ✓ Mode de synchronisation
  - ✓ Choix du convertisseur
  - ✓ Choix du validateur
  - ✓ Valeurs alternatives par défaut



GL Avancé: Prototypage et interfaces utilisateur

Data Binding avec NetBeans

- Conversion de type:
  - ✓ org.jdesktop.beansbinding.Convertor

```
public class TargetConverter extends Convertor<Source, Target> {

    public Target convertForward(Source source) {
            return Target.valueOf(source);
    }

    public Source convertReverse(Target target) {
            try {
                return Source.parseTarget(target);
            }
            catch (SourceFormatException e) {
                return null;
            }
    }
}
```

GL Avancé: Prototypage et interfaces utilisateur

Data Binding avec NetBeans

- Validation de saisie:
  - ✓ org.jdesktop.beansbinding.Validator

```
public class SourceValidator extends Validator<Source> {

    public Validator.Result validate(Source source) {
            if(source == null)
                return new Result(null, "Source must not be null");

            //Return a null result if source is valid
            return null;
    }
}
```

GL Avancé: Prototypage et interfaces utilisateur

Data Binding avec NetBeans

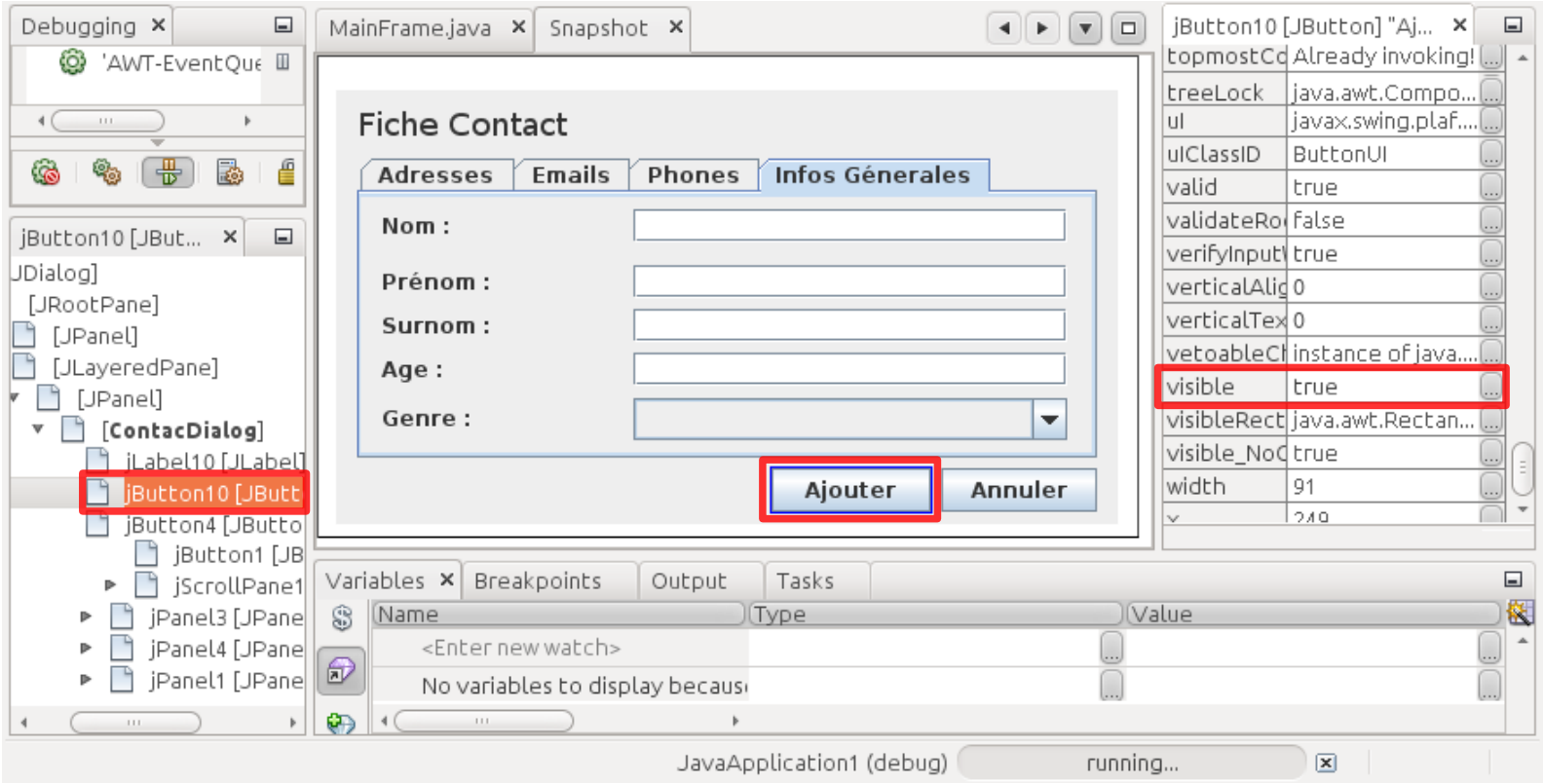
- Gestion du résultat de la validation par un écouteur
  - ✓ org.jdesktop.beansbinding.BindingListener

```
BindingListener bindingListener = new AbstractBindingListener() {
    public void synced(Binding binding) {
        JTextField textField = (JTextField)binding.getTargetObject();
        textField.setForeground(UIManager.getColor("Textfield.foreground"));
        textField.setToolTipText("");
    }

    public void syncFailed(Binding binding, SyncFailure failure) {
        JTextField textField = (JTextField)binding.getTargetObject();
        Validator.Result result = failure.getValidationResult();
        textField.setForeground(Color.red);
        textField.setToolTipText(result!=null?result.getDescription():"Sync failed");
    }
} ;
bindingGroup.addBindingListener( bindingListener );
```

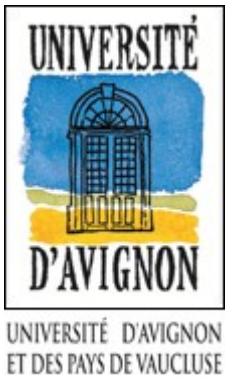
GL Avancé: Prototypage et interfaces utilisateur

Mise au point du prototype

- **Débogueur visuel (NetBeans 7.x)**
  - Capture d'écran de l'interface utilisateur en cours d'exécution
  - Navigation interactive dans les composants et consultation des valeurs des propriétés
  - Break point sur les composants, lien vers le code source, ...



Interactivité avec JavaFX



Master Informatique M1 2016-2017 - Université d'Avignon

Les propriétés en JavaFX

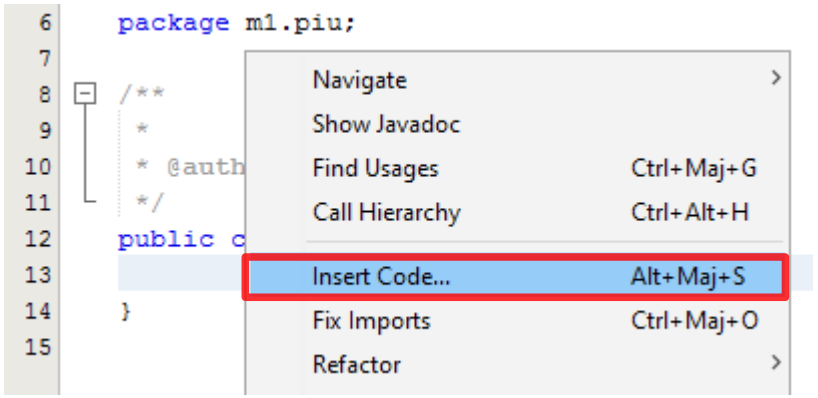
- JavaFX Properties : propriétés spécifiques pour les composants JavaFX
- Définies dans le package « javafx.beans.property »
- Améliore les conventions des propriétés JavaBeans (exemple pour une propriété « name »):

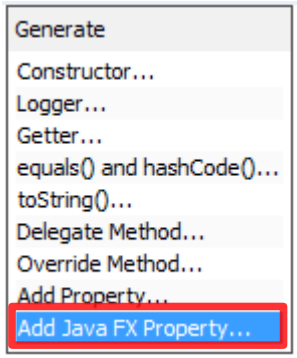
```
private StringProperty name = new SimpleStringProperty();

public String getName()  { return name.get();}

public void setName(String name){ this.name.set(name);}

public StringProperty nameProperty(){ return name;}
```

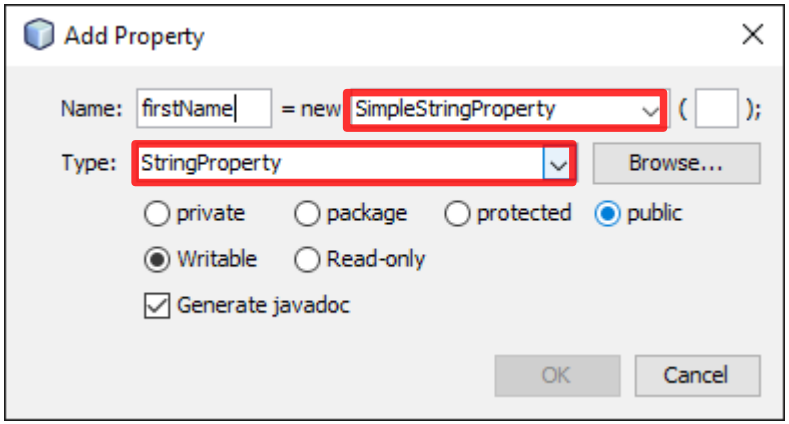
- Implémentation du pattern Observable :
  - ✓ Hérite de javafx.beans.Observable
  - ✓ Permet d'ajouter/retirer des écouteurs de changement (ChangeListener)

Les propriétés en JavaFX

- Génération automatique du code d'une propriété JavaFX dans NetBeans :
  - ✓ Menu contextuel depuis l'éditeur de code (ou depuis le menu « Source »)
  - ✓ Choix du type de propriété, et choix de l'implémentation








```
private StringProperty firstName = new SimpleStringProperty();

public String getFirstName()  { return firstName.get();}

public void setFirstName(String value){ this.firstName.set(value);}

public StringProperty firstNameProperty(){ return firstName;}
```

Le binding avec JavaFX

- JavaFX Binding : mise à jours automatiques des valeurs des propriétés liées
- Définies dans le package « javafx.beans.binding »
- Binding Haut niveau: accès simplifié facile avec une API de type « Fluent », et une classe « Bindings »

```
IntegerProperty num1 = new SimpleIntegerProperty(1);
IntegerProperty num2 = new SimpleIntegerProperty(2);

NumberBinding sum = Bindings.add(num1,num2); // Classe Bindings

NumberBinding  product = num1.multiply(num2); // API Fluent

System.out.println("Somme= "+sum.getValue())
System.out.println("Produit= "+product.getValue());
```

GL Avancé: Prototypage et interfaces utilisateur

Le binding avec JavaFX

- Binding Bas niveau: accès avancé plus performant avec extension des classes de bindings et surcharge de la méthode « computeValue() »

```
final DoubleProperty a = new SimpleDoubleProperty(1);
final DoubleProperty b = new SimpleDoubleProperty(2);
final DoubleProperty c = new SimpleDoubleProperty(3);
final DoubleProperty d = new SimpleDoubleProperty(4);

DoubleBinding db = new DoubleBinding() {

    {
        super.bind(a, b, c, d);
    }

    @Override
    protected double computeValue() {
        return (a.get() * b.get()) + (c.get() * d.get());
    }
};

System.out.println(db.get());
```

Les modèles de données avec JavaFX

- JavaFX Collections API est une extension du framework Collections de Java
- Définies dans le package « javafx.collections »
- Les interfaces proposées :
  - ObservableList: une liste qui permet de traquer les changements lorsqu'ils surviennent
  - ObservableMap: une map qui permet de traquer les changements lorsqu'ils surviennent
- Implémentation du pattern Observable
  - Hérite de javafx.beans.Observable
  - Permet d'ajouter/retirer des écouteurs de changement (ListChangeListener ou MapChangeListener)
- Classe utilitaire spécifique :
  - FXCollections: reprend les méthodes de java.util.Collections adaptées à JavaFX

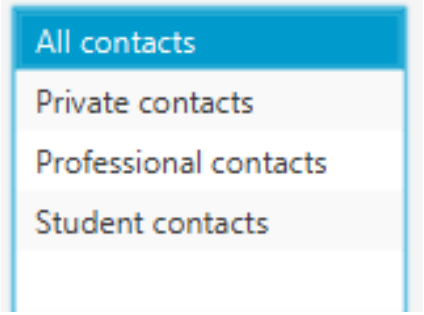
Les modèles de données avec JavaFX

- Peupler une liste en Java par le controller :
  - ✓ Créer une liste de valeurs
  - ✓ Initialiser le control JavaFX avec la liste

```
public void initialize(URL url, ResourceBundle rb) {
        ObservableList addresses = FXCollections.observableArrayList( "All contacts",
                [...],
                 "Student contacts");
        addressBooksListView.setItems(addresses);
}
```

- Peupler une liste en FXML :

```
<ListView fx:id="addressBooksListView" ...>
        <items>
                <FXCollections fx:factory="observableArrayList">
                        <String fx:value="All contacts"/>
                        [...]
                        <String fx:value="Student contacts"/>
                </FXCollections>
        </items>
</ListView>
```



Les modèles de données avec JavaFX

- Peupler un tableau en Java par le controller :
  - ✓ Créer une liste de valeurs avec des objets contenant des propriétés JavaFX
  - ✓ Initialiser le control JavaFX avec la liste
  - ✓ Définir les fabriques d'affichage des cellules

Nom	Prénom	Email
Titi 1	Toto 1	toto1.titi1@tata1.com
Titi 2	Toto 2	toto2.titi2@tata2.com
Titi 3	Toto 3	toto3.titi3@tata3.com

```
@FXML
private TableView<Contact> contactTable;
@FXML
private TableColumn<?, ?> contactLastNameColumn;
@FXML
private TableColumn<?, ?> contactFirstNameColumn;
@FXML
private TableColumn<?, ?> contactEmailColumn;

public void initialize(URL url, ResourceBundle rb) {
        ObservableList<Contact> contacts = FXCollections.observableArrayList(
                new Contact("Toto 1","Titi 1","toto1.titi1@tata1.com"));
        contactTable.setItems(contacts);

        contactLastNameColumn.setCellValueFactory(new PropertyValueFactory("lastName"));
        contactFirstNameColumn.setCellValueFactory(new PropertyValueFactory("firstName"));
        contactEmailColumn.setCellValueFactory(new PropertyValueFactory("email"));
}
```

GL Avancé: Prototypage et interfaces utilisateur

Les modèles de données avec JavaFX

- Peupler un tableau en FXML :

```
<TableView fx:id="contactTable" [...]>
    <columns>
        <TableColumn fx:id="contactLastNameColumn" [...] text="Nom" >
            <cellValueFactory><PropertyValueFactory property="firstName"/></cellValueFactory>
        </TableColumn>
        <TableColumn fx:id="contactFirstNameColumn" [...] text="Prénom" >
            <cellValueFactory><PropertyValueFactory property="lastName"/></cellValueFactory>
        </TableColumn>
        <TableColumn fx:id="contactEmailColumn" [...] text="Email" >
            <cellValueFactory><PropertyValueFactory property="email"/></cellValueFactory>
        </TableColumn>
    </columns>
    <items>
        <FXCollections fx:factory="observableArrayList">
            <Contact firstName="Toto 1" lastName="Titi 1" email="Email 1"/>
            <Contact firstName="Toto 2" lastName="Titi 2" email="Email 2"/>
        </FXCollections>
    </items>
    <columnResizePolicy>
        <TableView fx:constant="CONSTRAINED_RESIZE_POLICY" />
    </columnResizePolicy>
</TableView>
```

GL Avancé: Prototypage et interfaces utilisateur

Les modèles de données avec JavaFX

- Pagination d'un tableau avec JavaFX:

  - ✓ Navigation entre les pages d'un contenu divisé en sous ensembles

  - ✓ Indicateurs de page personnalisables : style class "STYLE_CLASS_BULLET"
  - ✓ Nombre d'indicateurs de pages : propriété "maxPageIndicatorCount"
  - ✓ Nombre d'indicateurs de pages modifiable par CSS : "-fx-max-page-indicator-count"

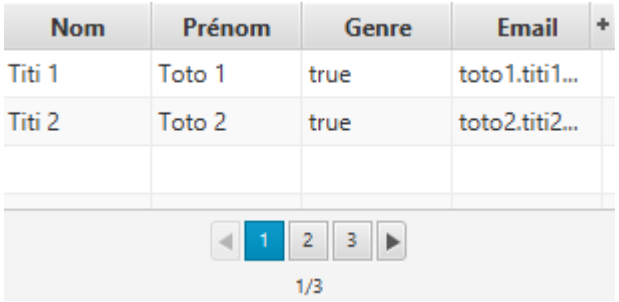
  - ✓ Nombre de pages déterminé par la propriété "pageCount" : (data.size() / rowsPerPage + 1)
  - ✓ Nombre de pages infini : "Pagination.INDETERMINATE"

  - ✓ Fonction callback appelée lorsqu'une page est sélectionnée
  - ✓ La fonction retourne le contenu, ou "Null" si la page n'existe pas

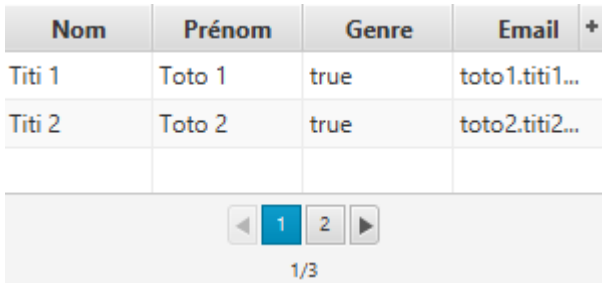
Les modèles de données avec JavaFX

- Pagination d'un tableau avec JavaFX:

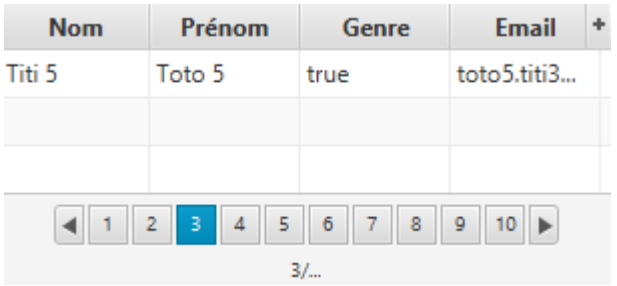
```
FXMLLoader loader = new FXMLLoader(getClass().getResource("FXMLContactTable.fxml"));
loader.load();
FXMLContactTableController controller = (FXMLContactTableController) loader.getController();
final TableView<Contact> contactTable = controller.getContactTable();

int rowsPerPage = 2;
Pagination pagination = new Pagination( (contacts.size() / rowsPerPage + 1), 0);

pagination.setPageFactory(pageIdx -> {
int fromIdx = pageIdx * rowsPerPage;
     int toIdx = Math.min(fromIdx + rowsPerPage, contacts.size());
     if(toIdx < fromIdx)
          return null;
     contactTable.setItems(FXCollections.observableArrayList(contacts.subList(fromIdx, toIdx)));
     return contactTable;
});
```



int rowsPerPage = 2



pagination.setMaxPageIndicatorCount(2)



Pagination.INDETERMINATE

Les modèles de données avec JavaFX

- Chargement infini d'un tableau avec JavaFX:
  - ✓ Pattern UI "Continuous Scrolling" ou "Infinite Scrolling"
  - ✓ Élimine le besoin de cliquer sur un bouton « Suivant »
  - ✓ Aucun arrêt dans le parcours des données, impression de contenu sans fin
  - ✓ Principe de fonctionnement en JavaFX :
    - Enregistrer un listener sur la ScrollBar pour tester si la position est au plus bas
    - Charger les éléments par lots et mettre à jours la position de la barre

Les modèles de données avec JavaFX

- Chargement infini d'un tableau avec JavaFX:

  - ✓ Méthode de création de contenu simulant le chargement:

```
private Contact generateContact(int nb) {
        return new Contact(
                String.format("Toto %1$02d",nb),
                String.format("Titi %1$02d",nb),
                String.format("toto%1$02d.titi%1$02d@tata%1$02d.com", nb));
}
```

  - ✓ Méthode de récupération de la ScrollBar du composant JavaFX:

```
private ScrollBar getTableViewScrollBar(TableView<?> view) {
    return view.lookupAll(".scroll-bar").stream()
            .filter(node -> node instanceof ScrollBar)
            .map(node -> (ScrollBar) node)
            .filter(bar -> bar.getOrientation().equals(Orientation.VERTICAL))
            .findFirst()
            .get();
}
```

Les modèles de données avec JavaFX

- Chargement infini d'un tableau avec JavaFX:
  - ✓ Méthode de chargement infini des données lors du défilement vers le bas:

```
final ScrollBar tableViewScrollBar = getTableViewScrollBar(contactTable);

tableViewScrollBar.valueProperty().addListener(
    (ObservableValue<? extends Number> observable, Number oldValue, Number newValue) -> {
        //Get current scroll bar position
        double position = newValue.doubleValue();

        //Check if position is bottom
        if (position != tableViewScrollBar.getMax())
            return;
        int step = contacts.size()+1;
        if (step <= maxContacts) {
            //Target position of scoll bar
            double targetPosition = position * contacts.size();
            //Find number of visible rows
            TableViewSkin<?> tvs = (TableViewSkin<?>) contactTable.getSkin();
            VirtualFlow<?> vf = (VirtualFlow<?>) tvs.getChildren().get(1);
            int visibleRow = vf.getLastVisibleCell().getIndex()-vf.getFirstVisibleCell().getIndex();
            //Generate nbVisibleRow new contacts and add to table
            contacts.addAll(IntStream.range(step, step + visibleRow)
                    .mapToObj(i -> generateContact(i))
                    .collect(Collectors.toList()));
            //Move position of scoll bar to allow further scroll to bottom
            getTableViewScrollBar(contactTable).setValue(targetPosition / contacts.size());
        }
    }
);
```

Bibliographie

- "Ergonomie du logiciel et design web", Jean François Nogier (Edition Dunod)
- "Critères Ergonomiques pour évaluer les IHM", Bastien, J. M. C., & Scapin, N° RT-0156 (1993)
  - http://hal.inria.fr/inria-00070012
- http://www.effectiveprototyping.com/
- http://www.lergonome.org/
- "100 NetBeans IDE Tips and Tricks", Ruth Kusterer
- http://netbeans.org/kb/articles/books.html

3ème partie:

Prototypage avancé