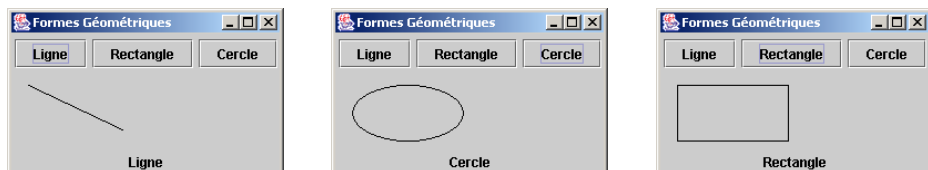


## TP JAVA n°8: Dessiner en Java - enum - Gestion des événements - JTable

### Exercice 1 : Dessiner en Java

1. Ecrire l'application graphique Swing vue en TD  
(taille conseillée pour la fenêtre largeur: 250 pixels Et hauteur: 150 pixels )



2. Modifier le code de votre application de manière à ce que la forme à dessiner (ligne, cercle, rectangle) **soit identifier à partir d'une énumération Forme**

### Exercice 2 : Implémentation du cas **Créer un patient** du use case **GererPatient** - Mise en place de l'interactivité dans la classe **PanelCreerPatient**

Dans un précédent TP, nous avons créé une application interactive qui proposait, entre autres, à un utilisateur de créer un Patient en **mode console**. Pour écrire cette application, nous avons utilisé le pattern MVC (**M**odèle **V**ue **C**ontrôleur). Rappelons que le **Contrôleur** (classe `GererPatientCtrl`) est en fait le **cœur de l'application** (*couche applicative*). Il assure l'interface avec l'utilisateur en faisant exécuter sa demande par la *couche métier* : la *couche présentation* n'a donc pas le droit d'attaquer directement la *couche métier*. Ainsi, pour que la **Vue** ne voit pas le **Modèle**, le **Contrôleur** sert de pont entre les deux. L'intérêt d'utiliser une telle architecture est que si la **Vue** change, nous n'avons pas besoin de ré-écrire le code lié aux parties **Modèle** (classes du paquetage `com.iut.cabinet.metier`) et **Contrôleur** (classes du paquetage `com.iut.cabinet.application`)... Seules les classes concernant la **Vue** (IHM) (c-a-d l'appel aux méthodes du contrôleur) doivent être réécrites... ce qui permet de changer facilement la **Vue** d'une application (ce que nous allons faire en exécutant maintenant notre application avec une **Interface Graphique Utilisateur**)

Dans un premier temps, nous nous intéresserons au **use case CréerPatient (sans ascendant)**.

Au cours du TP précédent, vous avez implémenté la classe `PanelCreerPatient` afin qu'elle propose l'interface ci-contre.

... si jamais, vous avez un problème avec le code de votre classe `PanelCreerPatient` écrite au TP précédent, vous pouvez utiliser pour ce TP la classe `PanelCreerPatient2` mis à votre disposition sur la zone libre, qui n'implémente pas la mise en forme demandée (vous chercherez la mise en forme plus tard...), mais vous permet de réaliser ce TP...

Il reste donc à mettre en place les événements en suivant la même démarche que la mise en place de l'application console c'est-à-dire en utilisant un **pattern DAO (Data Access Object)** qui facilitera une éventuelle "interchangeabilité" du support de persistance en **encapsulant les accès aux données** (la couche métier devient alors indépendante du support de persistance : pour ce TP, on travaille toujours

avec les fichiers, mais bientôt viendront les bases de données...)

Reprenons donc la démarche du « TP Tutoriel » pour la mise en place d'une application interactive en mode console (**MVC/DTO/DAO**) »

### ↳ **Mise en place du Contrôleur :**

Au lancement de l'application, vous devez mettre en place le modèle MVC en créant la **Vue**, mais aussi en créant le **Contrôleur**. Pour que la vue puisse envoyer des messages au contrôleur, il est nécessaire de reprendre **l'étape 2 du tutoriel Association du Contrôleur à la Vue utilisée**, et de déclarer et instancier dans la classe `PanelCreerPatient` le contrôleur (objet de type `GererPatientCtrl`) comme attribut de la **Vue** (classe `PanelCreerPatient`) comme vous l'aviez fait pour la classe `GererPatientIHM`.

### ↳ **Saisie des caractéristiques d'un Patient**

Lors de la mise en place de l'application console, les messages 2 et 3 du diagramme de séquence (voir annexe 2 du TP tutoriel) qui consistent à **saisir les caractéristiques du nouveau patient** étaient entremêlés dans la même méthode `creerPatient`.

Avec l'application graphique, ces deux étapes seront bien distinctes.

→ Tout d'abord **l'affichage de l'écran de saisie** consiste à proposer à l'utilisateur un formulaire à remplir avec tous les champs du **Patient** : c'est le code que vous avez écrit lors du TP précédent :

- soit vous l'avez directement écrit dans le constructeur de la classe `PanelCreerPatient`
- soit vous l'avez écrit dans une méthode `afficherEcranSaisie` qui est appelée par ce constructeur.

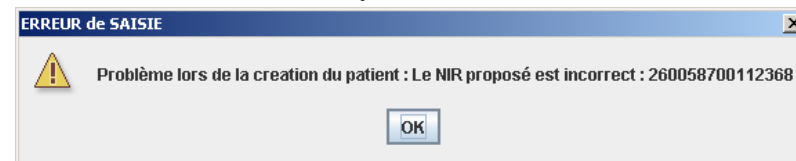
→ Puis la **validation de la saisie** va être possible récupérant un événement émis par le bouton `Valider`.

### ↳ **Gestion des événements pour le bouton Valider.**

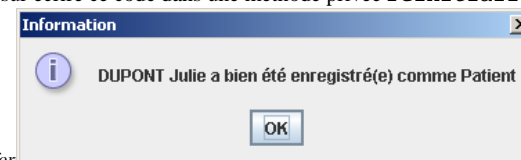
➤ Dans un premier temps, il est nécessaire d'**enregistrer un écouteur pour le bouton Valider**

➤ Le « comportement » du bouton `Valider` sera ensuite similaire au code déjà écrit à ce sujet dans la méthode `creerPatient` de la classe `GererPatientIHM` (lorsque nous avions une **Vue en mode console**) à savoir :

- récupération des données du formulaire (grâce à des `getText` et `getSelectedItem`)
- création d'un objet de type `PatientDTO` à partir des données du formulaire
- appel de la méthode `creerPatient` du contrôleur de use case.
  - Si quelque chose d'anormal se produit lors de la saisie, il faut informer l'utilisateur avec un message explicite (voir la classe `JOptionPane` dans la javadoc)



→ Si le patient a bien été enregistré, il faut afficher un message de prise en compte de l'enregistrement du patient (conformément au plan type détaillé), puis réinitialiser le formulaire (c-a-d « vider » tous les champs : les `JTextField` devront alors contenir des chaînes vides et le `JComboBox` devra présenter `Mr` : on peut bien sûr écrire ce code dans une méthode privée `reinitialiserFormulaire`).



### 🔗 Gestion des événements pour le bouton Quitter.

Enregistrer un écouteur pour le bouton **Quitter**.

Lorsque l'utilisateur clique sur le bouton **Quitter**, faites en sorte :

- que le panel ne soit plus visible et
- que le formulaire soit réinitialisé.

Pour tester le comportement du bouton **Quitter**, nous allons ajouter un objet de type

PanelCreerPatient sur la fenêtre CabMedMainFrame.

Créer dans le constructeur de la classe **CabMedMainFrame** un objet de type **PanelCreerPatient**, ajouter le sur la fenêtre et pour le moment, rendre cet objet non visible.

C'est maintenant le bon moment pour donner un comportement au menu **Créer un patient** de la barre de menu de classe CabMedMainFrame.

Lorsque ce menu est sélectionné, l'objet de type PanelCreerPatient devient visible...

Lancer l'application graphique et tester le comportement de votre bouton **Quitter**.

**Remarque :** La fenêtre principale n'est pas le seul conteneur possible pour « poser » un panel... La solution adoptée ici de rendre le panel visible ou non n'est sûrement pas la meilleure... (voir fin du TP dans la partie intitulée *en ce qui concerne l'affichage des panels*)

**Amélioration possible (à faire hors TP) :** Si l'utilisateur saisit des données incorrectes, vous pourriez par exemple faire en sorte que les champs erronés apparaissent en couleur rouge à l'écran ou que la zone de saisie soit colorée.. A vous de voir suivant votre inspiration ...

### Exercice 3 : Implémentation du cas **Lister tous les patients** du use case **GererPatient** à l'aide d'un composant **JTable**

Pour la **Vue** du use case **Lister tous les patients**, nous implémenterons dans le paquetage

com.iut.cabinet.presentation, une classe héritée de JPanel que nous appellerons

PanelListerPatients.

Dans ce panel, vous voulons que la **Vue** sur notre liste de Patients corresponde à la copie d'écran suivante :



Nom	Prenom	Date de Naiss...	NIR	Ascendant
DUPONT	Julie	21 mai 1960	260058700123...	<input type="checkbox"/>
DUPONT	Toto	25 déc. 1991	260058700123...	<input checked="" type="checkbox"/>
MARTIN	Jean	7 oct. 1968	168072B12345...	<input type="checkbox"/>

Pour obtenir un tel affichage, nous allons utiliser un **composant Swing « complexe »** appelé **JTable**.

Un composant **JTable** permet d'afficher des tables de données, en permettant éventuellement l'édition de ces données.

Son utilisation n'est pas triviale, c'est pourquoi dans un premier temps, nous allons nous familiariser avec ce composant en nous intéressant à son fonctionnement de base

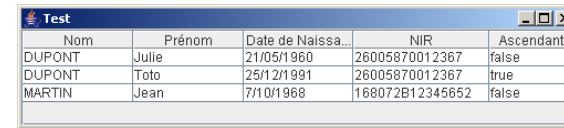
### 3.1 Manipulation d'un composant Swing « complexe » **JTable**

Pour comprendre le fonctionnement de base, commençons par étudier le code suivant qui correspond à la création d'une simple JTable.

```
////////////////////////////////////  
// Mise en place du Modèle de données à afficher  
////////////////////////////////////  
Object[][] data={  
    {"DUPONT", "Julie", "21/05/1960", "26005870012367", false},  
    {"DUPONT", "Toto", "25/12/1991", "26005870012367", true},  
    {"MARTIN", "Jean", "7/10/1968", "168072B12345652", false},  
};  
  
String[] columnNames={"Nom", "Prénom", "Date de Naissance", "NIR", "Ascendant"};  
  
////////////////////////////////////  
// Affichage la JTable  
// que l'on doit placer au préalable dans un conteneur de type JScrollPane  
////////////////////////////////////  
JTable maTable =new JTable(data, columnNames);  
JScrollPane monScrollPane = new JScrollPane(maTable);  
add(monScrollPane);
```

Récupérez ce code sur la zone libre en important la classe **PanelListerPatients** dans le paquetage **com.iut.cabinet.presentation**.

Exécutez la classe PanelListerPatients.



Nom	Prenom	Date de Naiss...	NIR	Ascendant
DUPONT	Julie	21/05/1960	26005870012367	false
DUPONT	Toto	25/12/1991	26005870012367	true
MARTIN	Jean	7/10/1968	168072B12345652	false

Le composant **JTable** vous permet déjà d'effectuer un certain nombre de manipulations sur le tableau affiché :

- modifier la largeur des colonnes (en plaçant le curseur de la souris sur l'en-tête entre deux colonnes)
- déplacer des colonnes (en cliquant sur l'en-tête d'une colonne, puis déplacer la souris tout en conservant le bouton enfoncé)
- éditer les cellules (double cliquer sur une cellule puis entrer des données au clavier)

Pour commencer, intéressons-nous à l'affichage de la **JTable**, et plus précisément aux instructions suivantes:

```
JScrollPane monScrollPane = new JScrollPane(maTable);  
add(monScrollPane);
```

Un composant **JTable** doit être ajouté à un composant **JScrollPane** (ascenseur) et non à un simple **JPanel**. Pour visualiser l'ascenseur, réduisez la taille de votre fenêtre d'exécution.

La **JTable** de l'exemple a été construite à partir du constructeur suivant :

```
JTable(Object[][] rowData, Object[] columnNames)  
Constructs a JTable to display the values in the two dimensional array, rowData, with column names, columnNames.
```

Ce constructeur accepte directement des données dans un tableau 2D.

Dans une première approche, **JTable** peut paraître simple à utiliser...

...mais avec ce constructeur, il y a quand même quelques petits détails gênants pour notre application :

- toutes les cellules sont éditables
- quel que soit le type de données, l'affichage est donné sous forme de **String** (comparez la colonne ascendant que vous venez d'obtenir avec la colonne ascendant de la copie d'écran de la page 3)
- et enfin, les cellules sont remplies avec les données d'un tableau 2D... or dans pour notre application, nous ne travaillons pas avec des tableaux 2D, mais bien avec des **Collection(s)**...

Pour éviter ce genre de problèmes, il faut passer par un modèle, et utiliser l'un des constructeurs suivants :

<b><code>JTable</code></b> ( <b><code>TableModel</code></b> dm)
Constructs a JTable that is initialized with dm as the data model, a default column model, and a default selection model.
<b><code>JTable</code></b> ( <b><code>TableModel</code></b> dm, <b><code>TableColumnModel</code></b> cm)
Constructs a JTable that is initialized with dm as the data model, cm as the column model, and a default selection model.
<b><code>JTable</code></b> ( <b><code>TableModel</code></b> dm, <b><code>TableColumnModel</code></b> cm, <b><code>ListSelectionModel</code></b> sm)
Constructs a JTable that is initialized with dm as the data model, cm as the column model, and sm as the selection model.

Ce qui veut dire qu'il faut fournir un objet `TableModel` au composant `JTable` afin de permettre à `JTable` de découvrir la valeur de chaque cellule. Autrement dit, `JTable` ne contient plus de données, mais est une **Vue** sur les données. En effet, tout composant `JTable` utilise un objet qui implémente `TableModel` (un Modèle) pour encapsuler les données qu'il visualise...

### 🔗 Création de son propre modèle de table :

Le modèle d'une `JTable` doit donc implémenter l'interface `TableModel`. Cette interface propose de nombreuses méthodes à redéfinir (voir javadoc et Annexe 1).

Pour écrire un modèle, on peut :

- 1. Ecrire une classe qui implémente directement `TableModel` (et donc implémenter toutes les méthodes de cette classe)
- 2. Ecrire une classe modèle qui hérite de `AbstractTableModel`. Cette classe abstraite implémente déjà l'interface `TableModel` à l'exception de trois méthodes :

`getRowCount`, `getColumnCount` et `getValueAt`

Pour une application nécessitant une table non éditable, il suffit d'implémenter cette classe anstraite

- 3. Utiliser la classe `DefaultTableModel` qui implémente toutes les méthodes de l'interface.

Dans la classe `DefaultTableModel`, les données sont stockées sous forme de vecteur de vecteurs (tableaux 2D)

### ➤ Solution n°1 : DefaultTableModel

Dans un premier temps, nous allons transformer le code de la classe `PanelListePatients` pour appeler un constructeur de la classe `JTable` nécessitant un objet de référence `TableModel` en argument. Nous commencerons avec une instance de `DefaultTableModel`.

Dans la classe `PanelListePatients`, remplacer la ligne :

```
JTable maTable =new JTable(data,columnNames);
```

par :

```
DefaultTableModel maTableModele=new DefaultTableModel(data,columnNames);  
JTable maTable =new JTable(maTableModele);
```

Exécuter et constater qu'on obtient le même affichage que précédemment. On pourrait bien sûr écrire une classe qui **hérite de `DefaultTableModel`** et **redéfinir les méthodes** dont l'implémentation ne nous convient pas, comme la méthode `isCellEditable` qui devrait renvoyer `false`, si on souhaitait que les cellules ne soient plus éditables. En plus, les données sont stockées sous forme de tableaux 2D, et nous souhaitons travailler directement avec des collections.

Nous allons donc préférer adopter la seconde solution et **créer un `TableModel` personnalisé qui hérite de `AbstractTableModel`**.

### ➤ Solution n°2 : TableModel personnalisé qui hérite de AbstractTableModel.

#### 🔗 Création de notre propre modèle de table :

→ Créer dans le paquetage `com.iut.cabinet.presentation` la classe `PatientDTOTableModel`.

La classe `PatientDTOTableModel` héritera de la classe `AbstractTableModel` du paquetage `javax.swing.table`.

→ Dans cette classe, vous commencerez par définir :

- 2 attributs :

```
private Collection<PatientDTO> data;  
private String[] columnNames = {"Nom", "Prenom",  
                                "Date de Naissance", "NIR", "Ascendant"};
```

- le constructeur suivant à 1 paramètre, qui permettra de personnaliser la collection :

```
public PatientDTOTableModel (Collection<PatientDTO> data) {  
    this.data = data;  
}
```

Pour être instanciable, cette classe devra obligatoirement redéfinir les trois méthodes abstraites de la classe `AbstractTableModel` à savoir :

```
public int getColumnCount() {...}  
public int getRowCount() {...}  
public Object getValueAt(int rowIndex, int columnIndex) {...}
```

**Remarque :** Eclipse, vous permet de générer automatiquement la déclaration de ces méthodes, de deux manières différentes

- soit en cliquant sur la croix rouge, après avoir écrit **extends** `AbstractTableModel`

```
public class PatientDTOTableModel extends AbstractTableModel{
```

- soit à partir du menu **Source**, puis en choisissant l'option **Override/Implements Method**

→ **Redéfinition de la méthode : `getColumnCount`** qui d'après la javadoc doit renvoyer le nombre de colonnes du modèle. Le nombre de colonnes correspond à la taille du tableau contenant le nom des colonnes donc :

```
public int getColumnCount() {  
    return columnNames.length;  
}
```

→ **Redéfinition de la méthode : `getRowCount`** qui d'après la javadoc doit renvoyer le nombre de lignes du modèle. Le nombre de lignes correspond donc à la taille de la collection :

```
public int getRowCount() {  
    if (this.data == null) return 0;  
    else return this.data.size();  
}
```

→ **Redéfinition de la méthode : `getValueAt`** qui d'après la javadoc doit renvoyer une instance de `Object` correspondant à la **valeur à afficher dans la cellule de la JTable** dont les indices des lignes et des colonnes sont passés en paramètre. Cette méthode est appelée par la `JTable` à chaque fois que le tableau doit être redessiné. Il est donc indispensable de coder cette méthode (si vous la laissez dans l'état actuel (return null), vous verrez une table vide) et de soigner son implémentation. Nous vous proposons l'implémentation suivante :

```
public Object getValueAt(int rowIndex, int columnIndex) {  
    PatientDTO pat = (PatientDTO) data.toArray()[rowIndex];  
    switch ( columnIndex ) {  
        case 0 : return pat.getNom();  
        case 1 : return pat.getPrenom();
```

```

        case 2 : return pat.getDateNaissance();
        case 3 : return pat.getNir();
        case 4 : if (pat.getUnAscendant()!=null) return true;
                  else return false;
        default : throw new IllegalArgumentException("Colonne Inconnue"
                                                    + columnIndex);
    }
}

```

→ En redéfinissant ces 3 méthodes, nous avons respecté le contrat de la classe abstraite mère `AbstractTableModel` et nous pouvons d'ores et déjà utiliser notre `PatientDTOTableModel` pour créer notre `JTable`.

Revenez dans la classe **PanelListerPatients** :

→ Effacer le code suivant

```

Object[][] data = .....
String[] columnNames= .....

```

... et remplacer-le par un appel au contrôleur de use case de `GererPatientCtrl` (avec le `try...catch` qui va bien...)

```
Collection<PatientDTO> maListe =ctrlUseCase.listerPatients();
```

→ Bien sûr, comme dans le cas de la création d'un patient en mode console, un objet `ctrlUseCase` de type `GererPatientCtrl` devra être déclaré au préalable comme attribut de la classe `PanelListerPatients` et correctement instancier :

```
GererPatientCtrl ctrlUseCase=new GererPatientCtrl();
```

→ Modifiez ensuite l'instanciation de votre objet `maTableModele`, qui doit être de type `PatientDTOTableModel` (et non plus `DefaultTableModel`) et prendre en paramètre du constructeur la collection précédemment récupérée.

```
PatientDTOTableModel maTableModele=new PatientDTOTableModel(maListe);
```

Exécutez...et testez ... Vous pouvez alors constater :

- 1. que les cellules ne sont plus éditables (aucune modification n'est maintenant permise)

→ Editer des cellules :

En fait, ceci est dû à l'implémentation dans la classe `AbstractTableModel` de la méthode

```
public boolean isCellEditable(int rowIndex, int columnIndex)
```

qui renvoie un booléen indiquant si la cellule située à la ligne `rowIndex` et à la colonne `columnIndex` est éditable. Pour vous en convaincre, vous pouvez redéfinir cette méthode de la manière suivante :

```

public boolean isCellEditable(int rowIndex, int columnIndex) {
    return true;
}

```

Remarque : pour obtenir facilement l'entête de la méthode à redéfinir, utiliser sous Eclipse le menu **Source** → **Override/Implements Method** et cocher la méthode à redéfinir

Exécutez et testez....

Toutes les cellules sont désormais éditables... Vous pouvez bien sûr choisir que seulement quelques cellules soit éditables en testant `rowIndex` et/ou `columnIndex` suivant les besoins de votre application.

Transformer le `return true` en `return false` afin que les cellules ne soient plus éditables...

- 2. que les colonnes sont nommées par des lettres A, B, C

→ Nommer les colonnes :

Pour nommer les colonnes avec vos propres entêtes, il est nécessaire de redéfinir la méthode `getColumnName` de la manière suivante :

```

public String getColumnName(int arg0) {
    return this.columnNames[arg0];
}

```

### → Utilisation du **Render** par défaut pour visualiser des cellules autrement que de façon standard :

Dans la copie d'écran présentée en début d'exercice, la valeur de la colonne **Ascendant** est représentée par une **JCheckBox** cochée si le patient possède un Ascendant. Dans les exécutions que nous avons obtenu jusqu'à présent toutes les cellules ont la même apparence, l'affichage est donné sous forme de texte aligné à gauche (comparez la colonne **Ascendant** que vous venez d'obtenir avec la colonne **Ascendant** de la copie d'écran de la page 3 de cet énoncé).

Il est possible de personnaliser l'affichage des cellules en créant ses propres objets **Render**, en implémentant une interface de type **Render**.

Si vous êtes intéressés, vous pourrez vous plonger (*hors TP*) dans la documentation officielle du composant **JTable**, un tutoriel Java « **How to Use Tables** » est disponible sur le site d'Oracle:

<http://download.oracle.com/javase/tutorial/uiswing/components/table.html>

... Mais si l'utilisateur ne souhaite pas créer ses propres **Render**(s), il peut quand même s'appuyer sur les **Render**(s) fournis par défaut. C'est ce que nous allons faire pour la suite du TP :

Configuration des **Render**(s) par défaut de la classe **JTable** :

Classe de l'objet à afficher	Composant renvoyé par le renderer	Alignement dans la cellule	Valeur affichée
Float ou Double	JLabel	Droite	Objet du modèle formaté avec une instance localisée de <b>NumberFormat</b>
Autres sous-classes de <b>Number</b>	JLabel	Droite	Résultat de l'appel à <b>toString</b> sur l'objet du modèle
<b>java.util.Date</b>	JLabel	Gauche	Objet du modèle formaté avec une instance localisée de <b>DateFormat</b>
<b>Boolean</b>	<b>JCheckBox</b>	Centré	Coché ou non suivant la valeur de l'objet du modèle
Icon ou <b>ImageIcon</b>	JLabel	Centré	Objet du modèle sous forme d'icône
Autres classes (dont classe <b>Object</b> )	<b>JLabel</b>	Gauche	Résultat de l'appel de <b>toString</b> sur l'objet du modèle

D'après ce tableau, nous devrions avoir pour notre booléen une **JCheckBox** cochée ou non, mais pourquoi n'est-ce pas encore le cas ?

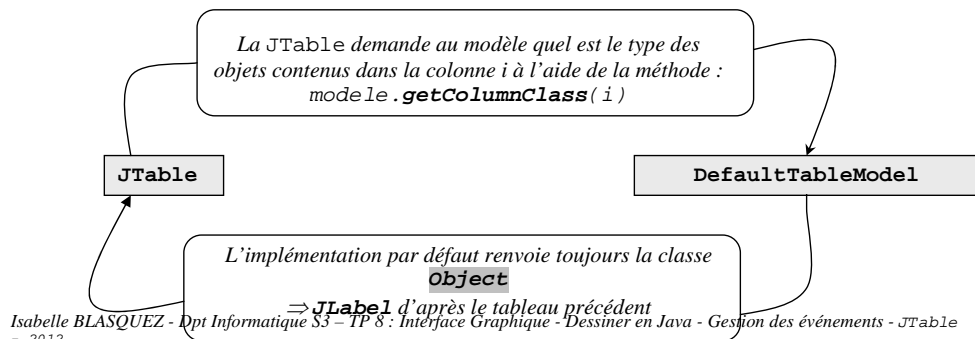
Pour comprendre cela, il faut s'intéresser à la mise en place des **renderers**...

Le tableau devrait connaître le type de la donnée à afficher (pour choisir le bon **render**...).

Or la **JTable** (**Vue**) ne dispose d'aucune information sur ces données, elle doit donc s'adresser à l'objet qui encapsule les données : le **TableModel**.

Comme nous l'avons vu précédemment, deux cas sont possibles :

→ **Cas n°1** : Il n'existe pas de **TableModel** créé par l'utilisateur, le composant **JTable** utilise alors un **TableModel** créé par défaut (**DefaultTableModel**).



→ **Cas n°2** : Un **TableModel** personnalisé a été implémenté, la méthode **getColumnClass** est consultée. Or, pour l'instant dans ce TP, nous avons uniquement codé les méthodes :

**getColumnCount**, **getRowCount**, **getValueAt** et **isCellEditable**

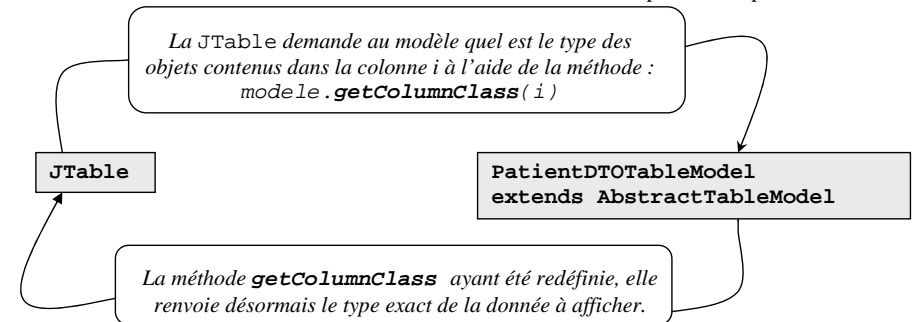
La méthode **getColumnClass** n'ayant pas été encore redéfinie, elle garde son implémentation par défaut, et nous renvoie toujours la classe **Object** **DefaultTableModel**.

La javadoc nous indique que la méthode **getColumnClass** (**int** columnIndex) doit renvoyer la classe des objet affichés dans la colonne columnIndex. Elle peut donc être implémentée de la manière suivante :

```
public Class<?> getColumnClass(int columnIndex)
{
    if (data == null) return Object.class;
    else
    {
        Object objetRecupere= getValueAt(0,columnIndex);
        if (objetRecupere==null) return Object.class;
        else return objetRecupere.getClass();
    }
}
```

Exécuter et tester....

Vos booléens ont désormais comme rendu une **JCheckBox** conformément à la première copie d'écran...



### → A propos de la sélection dans un **JTable**

Effectuer quelques tests de sélection sur votre **JTable**.

#### § Le Mode de sélection :

Actuellement, si vous cliquez dans votre **JTable**, vous sélectionnez **une ligne** du tableau(notre item) et vous pouvez effectuer **une sélection multiple**. La sélection multiple se fait de manière classique en utilisant :

- la touche « Shift » (Majuscule) pour sélectionner un intervalle
- ou la touche Ctrl pour sélectionner des lignes de façon discontinue

Il est possible de changer le mode de sélection en utilisant la méthode : **setCellSelectionEnabled** qui peut prendre comme constante :

**SINGLE\_SINGLETON** : un seul item peut être sélectionné

**SINGLE\_INTERVAL\_SELECTION** : un intervalle contigu d'items peut être sélectionné

**SINGLE\_INTERVAL\_SELECTION** : n'importe quelle combinaison d'items peut être sélectionnée : ce mode est le mode par défaut.

Pour la suite du TP, on souhaite ne permettre à l'utilisateur de ne sélectionner qu'une seule ligne, rajouter dans le constructeur de classe **PanelListePatients**, l'instruction suivante :

```
maTable.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```



### 🔗 *L'orientation de la sélection :*

Vous venez de le constater, le comportement standard d'un tableau est de permettre la sélection par ligne.

Pour information :

- Il est possible, si on le désire de changer l'orientation de la sélection et de permettre une sélection colonne par colonne. Pour cela il est nécessaire d'interdire la sélection des lignes et d'autoriser celles des colonnes :  
`maTable.setColumnSelectionAllowed(true);`  
`maTable.setRowSelectionAllowed(false);`
- Il est également possible d'autoriser une sélection cellule par cellule en utilisant le code suivant :  
`maTable.setCellSelectionEnabled(true);`

Pour la suite du TP, on restera sur le comportement suivant : sélection simple d'une ligne...

### 🔗 *Contenu de la sélection :*

Pour obtenir les indices de la ligne et de la colonne sélectionnée, nous pouvons utiliser les méthodes :

`getSelectedRow` et `getSelectedColumn`

🔗 *Création de modèle de sélection :* Sur le même principe que les `Renderers`, vous pouvez définir votre propre politique de sélection en implémentant l'interface `ListSelectionModel`. La sous-classe concrète utilisée par les composants standards est `DefaultListSelectionModel`

### ➔ Gestion des événements de la JTable

On souhaite mettre en place la fonctionnalité suivante :

Lorsque l'utilisateur *double clique* sur une ligne du tableau, un message d'information s'affiche détaillant complètement le patient (affichage de tous les attributs), comme l'indique la copie d'écran ci-contre.

Pour cela, vous devez enregistrer un écouteur de souris (`MouseListener`) auprès de votre `JTable` (`maTable`) et écrire un nouvel écouteur de type `MouseAdapter` qui n'aura besoin que de redéfinir la méthode `mouseClicked` de la manière suivante :

```
public void mouseClicked(MouseEvent e){
    // cas du double clic
    if(e.getClickCount()==2 &&
e.getSource() instanceof JTable){
        JTable maTable = (JTable)
e.getSource();
        int indiceLigne=maTable.getSelectedRow();
        // ... à vous de continuer le code
        // pour arriver au message d'information souhaité
        // ...
    }
}
```



**Remarque :** Pour détecter les changements de sélection (`ListSelectionListener`), il est possible d'abonner le listener soit au composant source (`JTable`), soit directement au modèle de sélection, si vous en avez créé un.

### ➔ Modification du modèle : pour information, non utilisé pour ce use-case

Il est possible de modifier le modèle de façon dynamique (ajout/suppression de lignes, modification de cellules), à chaque modification il faut informer la **V**ue des modifications apportées au **M**odèle en utilisant les méthodes préfixées par `fireTableXXX` de la classe `AbstractTableModel` qui permettent de décrire

finement la modification effectuée dans le modèle et d'optimiser la mise à jour effectuée à l'écran du tableau Swing (voir javadoc)

... ce qui doit absolument fonctionner :

- Créer un patient (flot de base)
- Lister tous les patients

... Toute amélioration et apport de nouvelles fonctionnalités seront bien évidemment prises en compte dans la notation finale...

#### Pour améliorer votre application :

➤ En vous inspirant de tout ce qui a été fait jusqu'à présent, continuez à programmer les différentes options de **GererPatient**. Afin de compléter les opérations du CRUD, il ne vous reste plus qu'à vous intéresser à ces deux parties.

⚡ **Implémentation du cas : Supprimer un Patient du use case GererPatient**

⚡ **Implémentation du cas : Modifier un Patient du use case GererPatient**

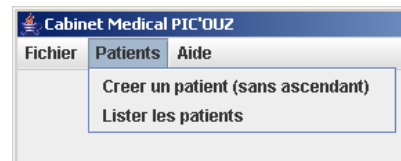
➤ Vous pouvez également vous intéresser à la **Prise en compte d'un ascendant** :

Pour créer un **Patient** ayant un ascendant, vous suivrez la démarche indiquée dans le plan type détaillée du « TP Tutoriel pour la mise en place d'une application interactive en mode console (MVC/DTO/DAO) »

➤ Vous pouvez également vous intéresser aux **Professionnels**.

➤ En ce qui concerne l'affichage des panels ...

...Nous voulons respecter la contrainte suivante :  
Lorsque l'utilisateur a « choisi » une option du menu **Patients**, et que le panel correspondant est affiché à l'écran, nous souhaitons que l'utilisateur ne puisse pas ouvrir un autre panel avant d'avoir quitté celui sur lequel il est :



Plusieurs solutions et plusieurs pistes sont envisageables...

( CardLayout, JInternalFrame, JtabbedPane, jouer sur le blocage des options du menu, autre solution, etc ...)

A vous de voir laquelle vous convient le mieux .....

➤ A propos du use case Lister tous les Patients ...

Vous pouvez améliorer la recherche d'un patient en proposant différentes *options de recherche avancée* comme par exemple :

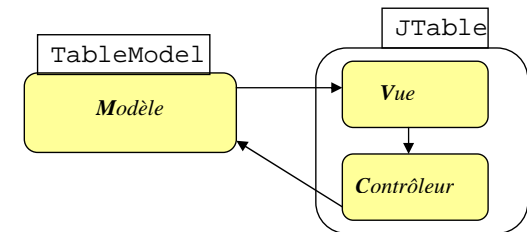
- en proposant une recherche « sélective » qui peut être mise en place en rajoutant dans le panel, au-dessus de du tableau `JTable` une zone de texte qui permet de saisir le nom du patient rechercher. On pourrait imaginer qu'à chaque fois qu'un caractère est tapé dans cette zone texte, la table est mise à jour pour n'afficher que les patient dont le nom commence par le texte saisi (Par exemple : si DU est tapé, on affichera DUPONT, DURAND, etc...)
- en proposant de trier la table suivant différents critères, etc ...
- ... à vous d'imaginer d'autres fonctionnalités....

#### Un résumé sur JTable :

Quand vous créez votre propre modèle, vous devez écrire une classe qui hérite de `AbstractTableModel` et qui doit :

- obligatoirement implémenter les 3 méthodes `getColumnCount`, `getRowCount` et `getValueAt`
- éventuellement redéfinir les autres méthodes de l'interface `TableModel` dont le comportement ne vous convient pas ...

En fait, l'implémentation du composant Swing `JTable` proposé par JAVA respecte le principe de l'architecture MVC (où la Vue et le Contrôleur ne sont pas dissociés)

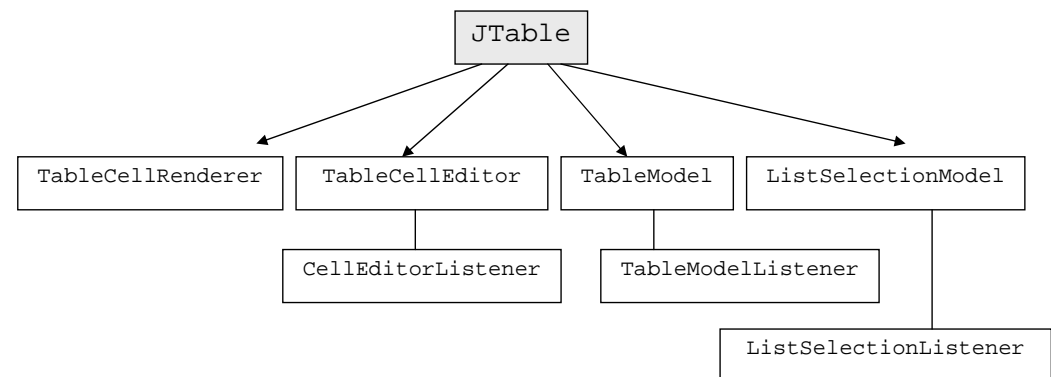


La classe `JTable` est la classe de Swing qui offre le plus de fonctionnalités.

En plus de l'implémentation de l'interface `TableModel`, il est possible de personnaliser sa `JTable` en utilisant des classes spécifiques pour effectuer :

- **un rendu des cellules** à l'aide d'un objet personnalisé de type **Renderer** qui retourne une visualisation de la données dans la cellule (Dans le cadre de ce TP, nous avons juste utiliser le `Renderer` par défaut, mais il est possible de définir ses propres `Renderer`).
- **une édition des données de cellules** à l'aide d'un objet **Editor** qui peut être appelée quand l'utilisateur effectue une action sur la cellule (à condition que la cellule soit éditable)
- **une définition d'un modèle de sélection** propre à la `JTable` en implémentant l'interface `ListSelectionModel`. La définition d'un nouveau modèle de sélection n'est utile que si vous souhaitez effectuer des traitements « pointus » des sélections : on s'en est très bien sorti avec un `MouseListener`.

Le composant `JTable` est donc une **Vue paramétrable** qui permet d'afficher ou de modifier des données d'une table en utilisant les nombreuses classes liées à ce composant Swing.



L'utilisation de `JTable` est très complexe et permet de nombreuses fonctionnalités...

si vous voulez en savoir un peu plus, consulter la documentation de Sun à ce sujet :

<http://download.oracle.com/javase/tutorial/uiswing/components/table.html>

et les nombreux tutoriaux que vous trouverez sur le web ...

## Annexe 1 : Interface TableModel

### javax.swing.table Interface TableModel

All Known Implementing Classes:  
[AbstractTableModel](#), [DefaultTableModel](#)

public interface **TableModel**

The TableModel interface specifies the methods the JTable will use to interrogate a tabular data model.

The JTable can be set up to display any data model which implements the TableModel interface with a couple of lines of code:

```
TableModel myData = new MyTableModel();  
JTable table = new JTable(myData);
```

For further documentation, see [Creating a Table Model](#) in *The Java Tutorial*.

See Also:  
[JTable](#)

### Method Summary

void	<a href="#">addTableModelListener</a> ( <a href="#">TableModelListener</a> l)	Adds a listener to the list that is notified each time a change to the data model occurs.
<a href="#">Class</a> < <a href="#">T</a> >	<a href="#">getColumnClass</a> (int columnIndex)	Returns the most specific superclass for all the cell values in the column.
int	<a href="#">getColumnCount</a> ()	Returns the number of columns in the model.
<a href="#">String</a>	<a href="#">getColumnName</a> (int columnIndex)	Returns the name of the column at columnIndex.
int	<a href="#">getRowCount</a> ()	Returns the number of rows in the model.
<a href="#">Object</a>	<a href="#">getValueAt</a> (int rowIndex, int columnIndex)	Returns the value for the cell at columnIndex and rowIndex.
boolean	<a href="#">isCellEditable</a> (int rowIndex, int columnIndex)	Returns true if the cell at rowIndex and columnIndex is editable.
void	<a href="#">removeTableModelListener</a> ( <a href="#">TableModelListener</a> l)	Removes a listener from the list that is notified each time a change to the data model occurs.
void	<a href="#">setValueAt</a> ( <a href="#">Object</a> aValue, int rowIndex, int columnIndex)	Sets the value in the cell at columnIndex and rowIndex to aValue.

Interface  
**TableModel**

**AbstractTableModel**

**DefaultTableModel**

### javax.swing.table Class AbstractTableModel

[java.lang.Object](#)  
└─ [javax.swing.table.AbstractTableModel](#)

All Implemented Interfaces:  
[Serializable](#), [TableModel](#)

Direct Known Subclasses:  
[DefaultTableModel](#)

```
public abstract class AbstractTableModel  
extends Object  
implements TableModel, Serializable
```

This abstract class provides default implementations for most of the methods in the TableModel interface. It takes care of the management of listeners and provides some conveniences for generating TableModelEvents and dispatching them to the listeners. To create a concrete TableModel as a subclass of AbstractTableModel you need only provide implementations for the following three methods:

```
public int getRowCount();  
public int getColumnCount();  
public Object getValueAt(int row, int column);
```

### javax.swing.table Class DefaultTableModel

[java.lang.Object](#)  
└─ [javax.swing.table.AbstractTableModel](#)  
└─ [javax.swing.table.DefaultTableModel](#)

All Implemented Interfaces:  
[Serializable](#), [TableModel](#)

```
public class DefaultTableModel  
extends AbstractTableModel  
implements Serializable
```

This is an implementation of TableModel that uses a Vector of Vectors to store the cell value objects.

## Annexe 2 : Comment accéder facilement aux tutoriels proposés sur le site d'Oracle depuis la javadoc d'une classe donnée ?

Pour retrouver le tutoriel officiel sur la manipulation de JTable, vous pouvez ouvrir la javadoc de la classe JTable et cliquer sur le lien :

[How to Use Tables](#)

### javax.swing Class JTable

[java.lang.Object](#)  
└─ [java.awt.Component](#)  
└─ [java.awt.Container](#)  
└─ [javax.swing.JComponent](#)  
└─ [javax.swing.JTable](#)

All Implemented Interfaces:

[ImageObserver](#), [MenuContainer](#), [Serializable](#), [EventListener](#), [Accessible](#), [CellEditorListener](#), [ListSelectionListener](#), [RowSorterListener](#), [TableColumnModelListener](#), [TableModelListener](#), [Scrollable](#)

```
public class JTable  
extends JComponent  
implements TableModelListener, Scrollable, TableColumnModelListener, ListSelectionListener, Cell
```

The JTable is used to display and edit regular two-dimensional tables of cells. See [How to Use Tables](#) in *The Java Tutorial* for task-oriented documentation and examples of using JTable.

Garder cette démarche à l'esprit, et soyez curieux en consultant la javadoc officielle...  
De nombreux liens vers des tutoriels sont proposés dans les classes de l'API standard...