Les Tables

- Le Composant JTable
- TableModel & AbstractTableModel
- ColumnModel & TableColumn
- Gestion des évènements
- Gestion de la sélection

Les Tables

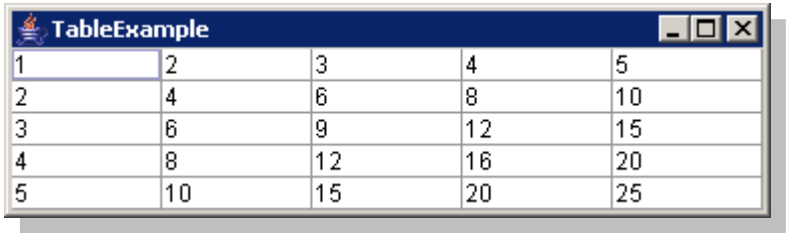
- La **`JTable`** est un composant affichant/éditant des données sous forme de feuille de calcul.
- Vision d'ensemble : six autres classes/interfaces utilisées :
  - **`TableModel`** : indique les données figurant dans la table
  - **`TableColumnModel`** : modèle pour les colonnes de la table
  - **`JTableHeader`** : composant graphique gérant les titres des colonnes.
  - **`TableSelectionModel`** : modèle de sélection des noeuds
  - **`TableCellRenderer`** : interface de rendu d'un noeud
  - **`TableCellEditor`** : l'éditeur de noeuds
- Elles sont définies dans **`javax.swing.table.*`**

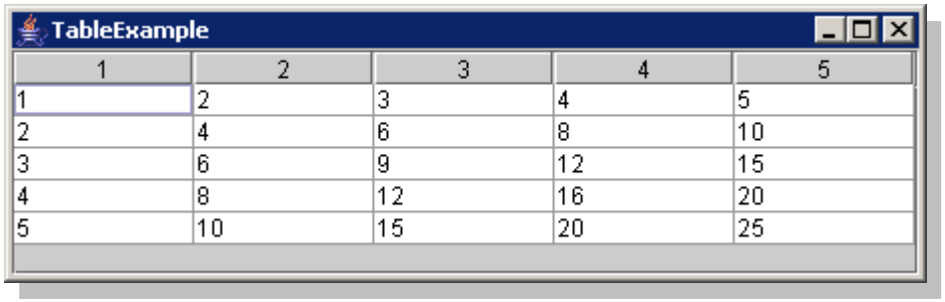
Construire une table

- Différentes façon de construire une table :
  - **`JTable(TableModel dm, TableColumnModel cm, ListSelectionModel sm)`**
    Les trois modèles sont fournis.
  - **`JTable(TableModel dm)`**
    avec le modèle de données **`dm`**, les autres par défaut.
  - **`JTable(TableModel dm, TableColumnModel cm)`**
    avec modèle de données et modèle de colonnes fournis.
  - **`JTable(int numRows, int numColumns)`**
    Toutes les cellules sont vides.
  - **`JTable(Object[][] rowData, Object[] columnNames)`**
    avec les valeurs des cellules de **`rowData`** et
    noms de colonnes **`columnNames`**.

JTable & JTableHeader

- Une table n'affiche pas par défaut de titre pour ses colonnes.
- Le composant graphique **`JTableHeader`** représente l'ensemble des en-têtes de colonne.





- Il faut mettre la **`JTable`** dans un **`JScrollPane`** pour que le **`JTableHeader`** soit placé comme *header* du JScrollPane.

Modèle de table

- Une table est créée à partir d'un **`TableModel`**
  - Nombre de lignes

```
int getRowCount()
```

  - Nombre de colonnes

```
int getColumnCount()
```

  - Obtenir la valeur d'une cellule

```
Object getValueAt(int row, int column)
```

  - Obtenir le nom d'une colonne

```
String getColumnName(int column)
```

  - Obtenir le type des valeurs pour une colonne

```
Class<?> getColumnClass(int column)
```

Modèle de table (2)

- Modification des cellules
  - Indiquer que des cellules sont éditables

```
boolean isCellEditable(int row, int column)
```

  - Changer la valeur des données

```
void setValueAt(Object value, int row, int column)
```

- Gestion des évènement de données
  - Ajouter un écouteur de modification de données

```
void  addTableModelListener(TableModelListener l)
```

  - Retirer un écouteur

```
void  removeTableModelListener(TableModelListener l)
```

Modèle abstrait d'arbre

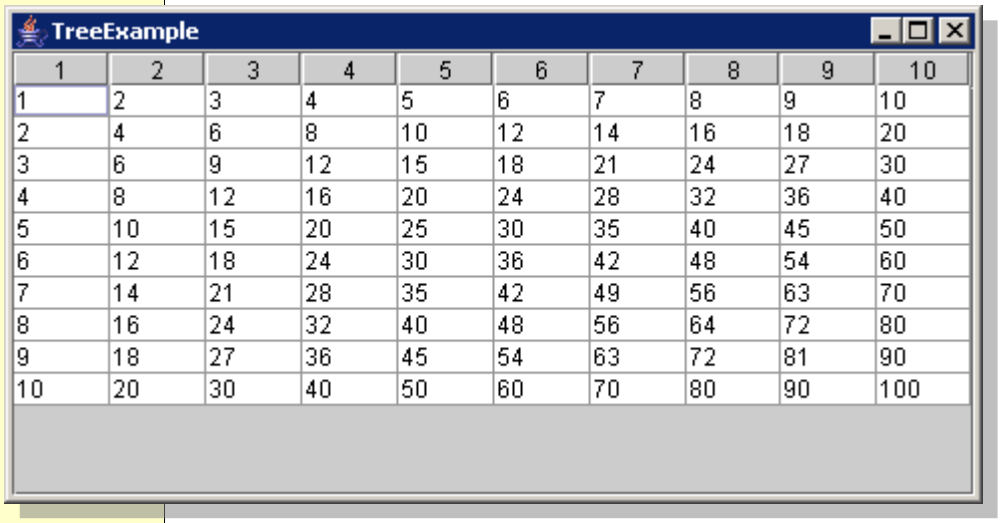
- La classe abstraite **`AbstractTableModel`**implante les méthodes :
  - getColumnName() renvoie 'A', 'B', etc.
  - getColumnClass() renvoie Object.class.
  - isCellEditable() renvoie toujours faux.
  - setValueAt() ne fait rien.
  - Plus la gestion des évènements (add...Listener, fire*).

- Reste à implanter les méthodes :
  - getRowCount() : le nombre de lignes
  - getColumnCount() : le nombre de colonnes
  - getValueAt(int row,int column) : la valeur de la cellule

Exemple simple de modèle de table

- Affiche une table de multiplication.

```
class MultTableModel extends AbstractTableModel {
  public int getColumnCount() {
    return 10;
  }
  public int getRowCount() {
    return 10;
  }
  public Integer getValueAt(int row, int column) {
    return (row+1)*(column+1);
  }
  public String getColumnName(int column) {
    return Integer.toString(column+1);
  }
}
..
public static void main(String[] args) {
  TableModel model=new MultTableModel();
  JTable table=new JTable(model);

  JFrame frame=new JFrame("TreeExample");
  frame.setContentPane(new JScrollPane(table));
  frame.pack();
  frame.setVisible(true);
}
```



1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Propriétés des fichiers

- Affiche les propriétés des fichiers dans une table.

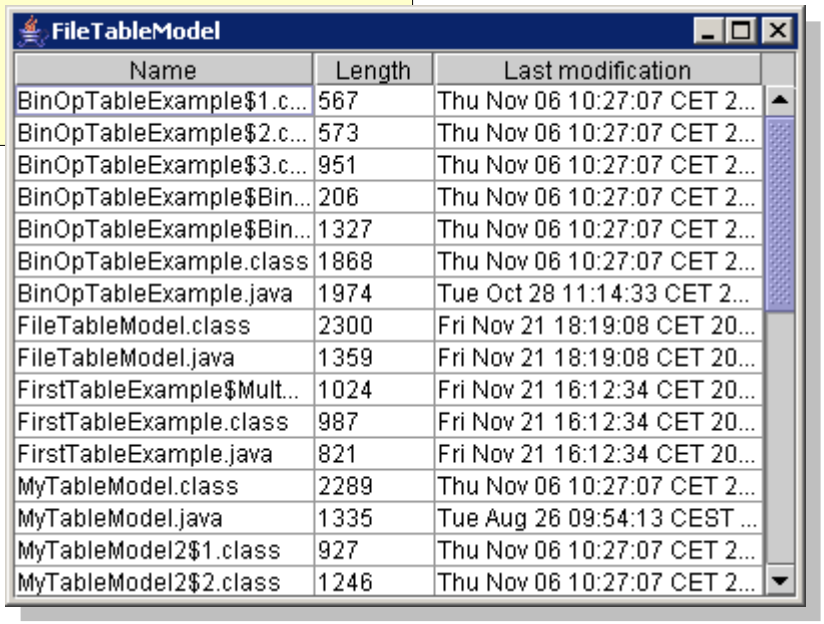
```
public class FileTableModel extends AbstractTableModel {
  public FileTableModel(File directory) {
    files=directory.listFiles();
  }
  public int getColumnCount() {
    return 3;
  }
  public int getRowCount() {
    return files.length;
  }
  public String getColumnName(int column) {
    return columnNames[column];
  }
```

```
  public Object getValueAt(int row, int column) {
    File file=files[row];
    switch(column) {
      case 0:
        return file.getName();
      case 1:
        return file.length(); // boxing
      case 2:
        return new Date(file.lastModified());
    }
    throw new AssertionError(
      "invalid column ("+row+','+column+')');
  }

  private final File[] files;
  private final static String[] columnNames={
    "Name","Length","Last modification"
  };
```

Propriétés des fichiers (2)

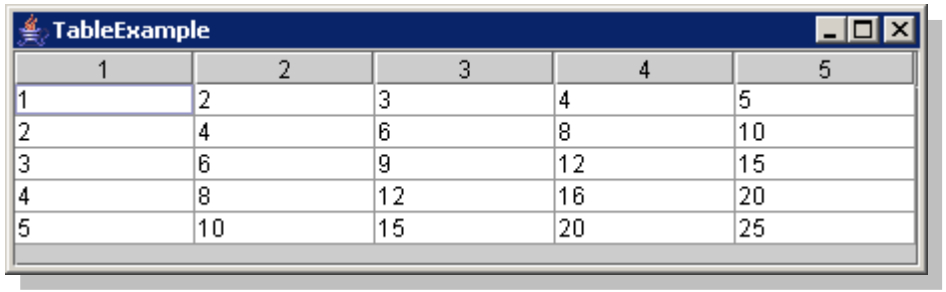
- Affiche les propriétés des fichiers dans une table.

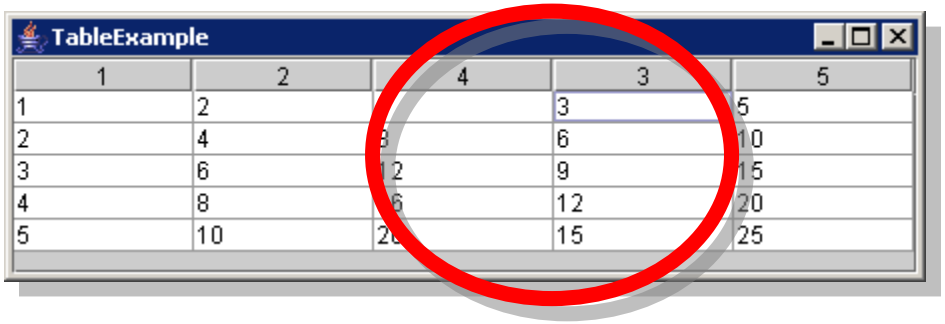
```
public static void main(String[] args) {
  FileTableModel model=new FileTableModel(new File("."));

  JTable table=new JTable(model);
  JScrollPane pane=new JScrollPane(table);

  JFrame frame=new JFrame();
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setContentPane(pane);
  frame.setSize(400,300);
  frame.setVisible(true);
}
```



Gestion des colonnes

- La JTable délègue au modèle de colonne **`ColumnModel`** la gestion des colonnes.
- Le **`ColumnModel`** gère une indirection graphique entre l'index d'une colonne dans la vue (viewIndex) et l'index de la colonne dans le modèle (modelIndex).
- Cette indirection permet à un utilisateur de déplacer les colonnes.





Gestion des colonnes (2)

- Le **`ColumnModel`** utilise des objets **`TableColumn`** pour représenter les colonnes.
- La méthode **`getColumn(int viewIndex)`** associe un **`TableColumn`** à un index dans la vue.
- Une colonne (**`TableColumn`**) possède :
  - Un index (modelIndex) dans le modèle de données.
  - Une taille (width)
  - Un *renderer* (cellRenderer) qui peut être null.
  - Un *editor* (cellEditor) qui peut être null.

```
TableColumn(int modelIndex, int width,
 TableCellRenderer renderer, TableCellEditor editor)
```

Gestion des colonnes (3)

- Deux méthodes dans **`JTable`** permettent de faire la convertion entre un index de la vue et celui du modèle
  - int convertColumnIndexToModel(int viewIndex)
  - int convertColumnIndexToView(int modelIndex)

- Le code de **`convertColumnIndexToModel()`**est :

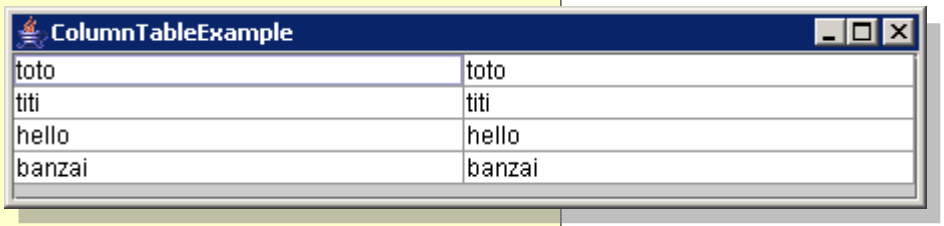
```
getColumnModel().getColumn(viewIndex).getModelIndex()
```

- **`viewIndex`**=2
  `modelIndex`=3

Exemple d'utilisation des colonnes

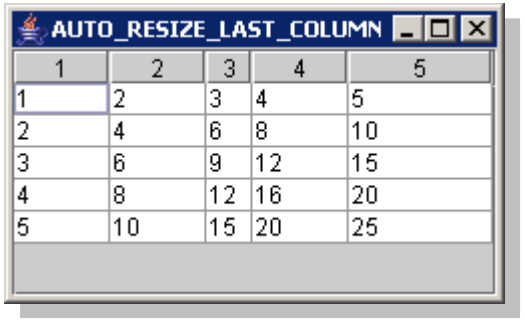
- Deux colonnes avec les mêmes données.

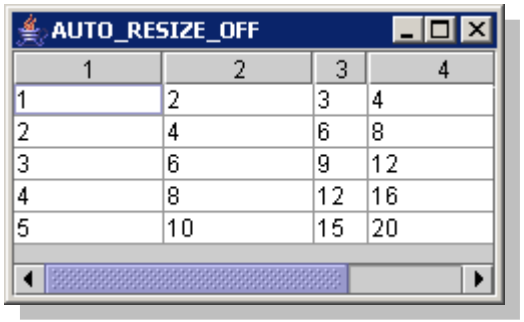
```
public static void main(final String[] args) {
  TableModel dataModel=new AbstractTableModel() {
    public int getColumnCount() {
      return 1;
    }
    public int getRowCount() {
      return args.length;
    }
    public String getValueAt(int row, int column) {
      return args[row];
    }
  };

  TableColumnModel columnModel=new DefaultTableColumnModel();
  TableColumn column1=new TableColumn(0,100);
  columnModel.addColumn(column1);
  TableColumn column2=new TableColumn(0,100);
  columnModel.addColumn(column2);

  JTable table=new JTable(dataModel,columnModel);
  ...
}
```

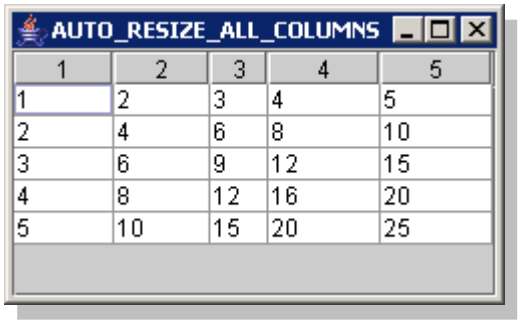


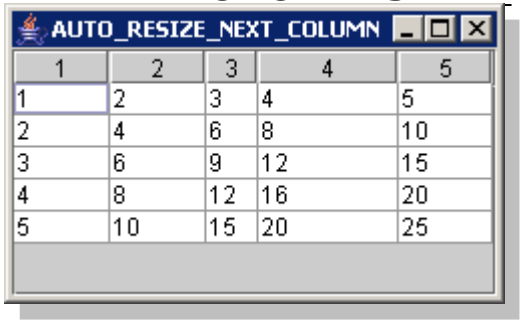
Taille des colonnes

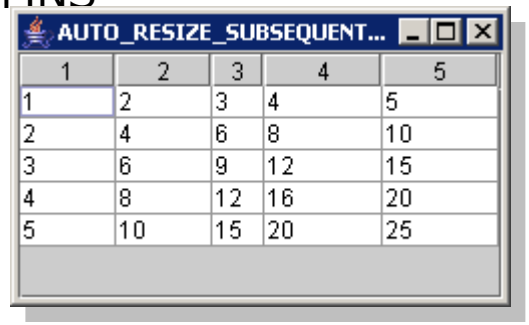
- Lors du changement de de taille d'une colonne : **`setAutoResizeMode(int mode)`**
  - Pas de resize : AUTO_RESIZE_OFF
  - Change la taille de la colonne suivante :AUTO_RESIZE_NEXT_COLUMN
  - Change les tailles des colonnes suivantes :AUTO_RESIZE_SUBSEQUENT_COLUMNS
  - Change la taille de la dernière colonne : AUTO_RESIZE_LAST_COLUMN
  - Change toutes les autres colonnes : AUTO_RESIZE_ALL_COLUMNS











Changer le composant de rendu

- Le **TableCellRenderer** permet de changer l'affichage graphique de chaque cellule.

- Deux solutions pour indiquer un *renderer* :
  - Sur une colonne :

```
table.getColumnModel().getColumn(viewIndex).
  setCellRenderer(TableCellRenderer r)
```

  - Pour un type de données :

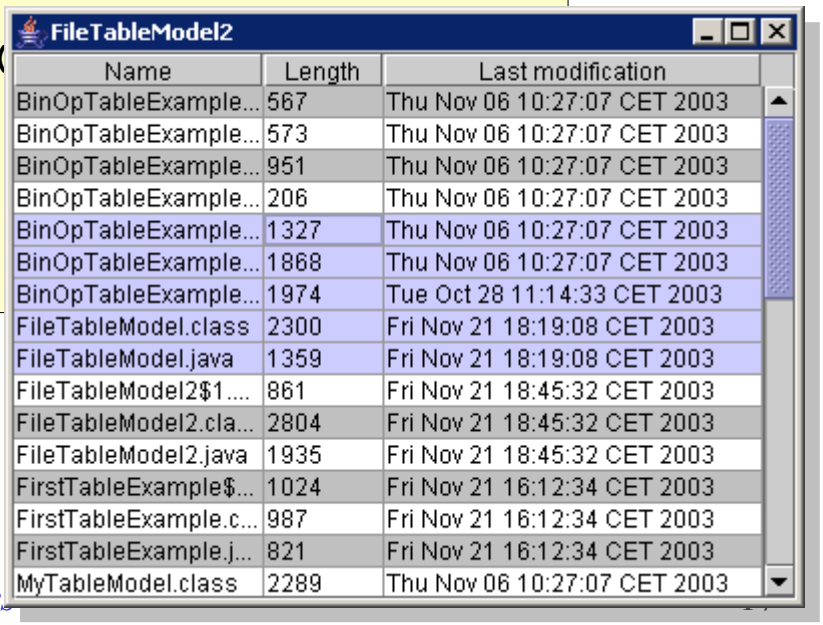
```
table.setDefaultRenderer(Class type,TableCellRenderer r)
```

- La classe **DefaultTableCellRenderer** implante le *renderer* par défaut.

Afficher des lignes de couleurs différentes

- Redéfinie la méthode **getTableCellRendererComponent()** du **DefaultTableCellRenderer**

```
public static void main(String[] args) {
  FileJTable table=new JTable(model);
  table.setDefaultRenderer(Object.class, new DefaultTableCellRenderer(){
    public Component getTableCellRendererComponent(
      JTable table,Object value,boolean isSelected,
      boolean hasFocus,int row,int column) {

      setBackground((row%2==0)?Color.LIGHT_GRAY:Color.WHITE);

      return super.getTableCellRendererComponent(
        table,value,isSelected,hasFocus,
        row,column);
    }
  });
}
```

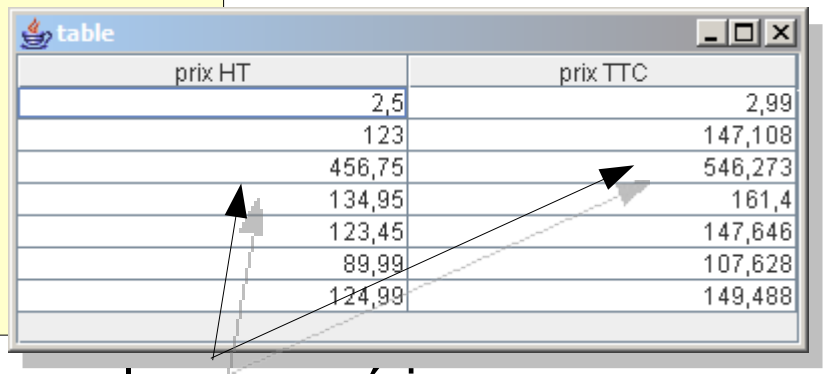


Exemple de celulles avec des valeurs numériques

- Le renderer par défaut dépend du type.

```
public class MutableTableModel extends AbstractTableModel {
  public MutableTableModel(double... prices) {
    this.prices=prices;
  }
  public int getColumnCount() {
    return 2;
  }
  public int getRowCount() {
    return prices.length;
  }
  public String getColumnName(int column) {
    return (column==0)?"prix HT":"prix TTC";
  }
  public Class<Double> getColumnClass(int columnIndex) {
    return Double.class;
  }
  public Double getValueAt(int row, int column) {
    double price=prices[row];
    return (column==0)?price:price*1.196;
  }
  private final double[] prices;
```

```
  public static void main(String[] args) {
    MutableTableModel model=new MutableTableModel(
      2.50, 123.0, 456.75, 134.95,
      123.45, 89.99, 124.99);
    ...
  }
```



Renderer par défaut des valeurs numériques

Edition des cellules

- Au niveau du **TableModel** :
  - indiquer les cellules éditables :

```
boolean isEditable(int row, int column)
```

  - Indiquer comment changer la valeur de la cellule

```
void setValueAt(Object value, int row,int column)
```

- Specifier, si besoin un editor :

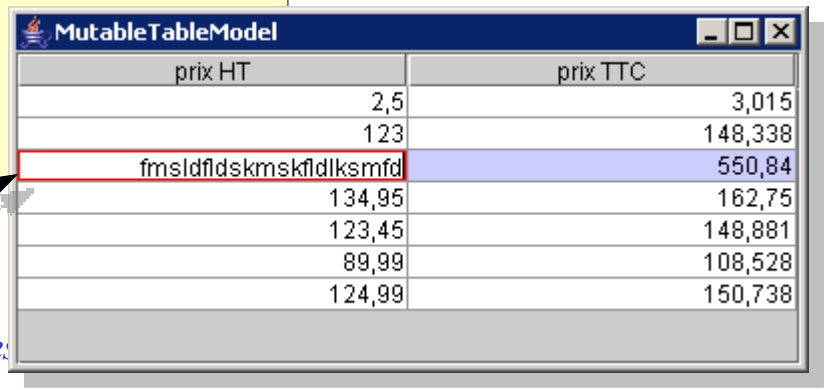
```
table.getColumnModel().getColumn(viewIndex).
  setCellEditor(TableCellEditor editor)
```

```
table.setDefaultEditor(Class type,TableCellEditor editor)
```

Exemple d'édition des celulles

- **`isEditable()`** permet d'indiquer les cellules éditables.
- L'éditeur par défaut dépend aussi du type.

```
public class MutableTableModel extends AbstractTableModel {
  public MutableTableModel(double[] prices) {
    this.prices=prices;
  }
  ...
  public boolean isCellEditable(int row, int column) {
    return column==0;
  }
  public void setValueAt(Object value, int row, int column) {
    prices[row]=(Double)value;
    fireTableRowsUpdated(row, row);
  }
  public Class<Double> getColumnClass(int columnIndex) {
    return Double.class;
  }
  private final double[] prices;
}
```



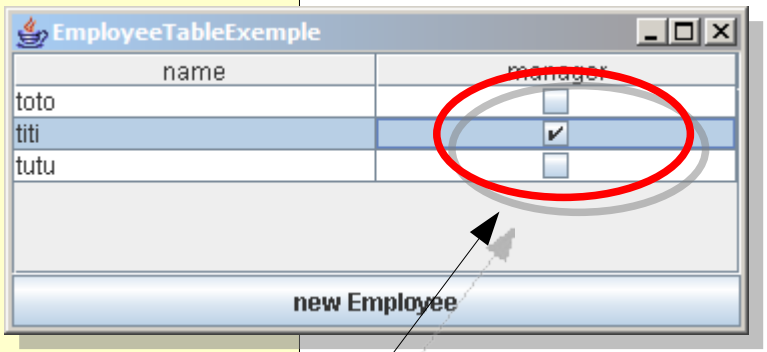
Editor par défaut
des valeurs numériques

Rendu des valeurs booléennes

- Le renderer par défaut est une case à cocher

```
public static class EmployeeTableModel extends AbstractTableModel {
  ...
  public Object getValueAt(int row, int column) {
    Employee employee=employees.get(row);
    return (column==0)?employee.getName():employee.isManager();
  }
  public Class<?> getColumnClass(int column) {
    return (column==0)?String.class:Boolean.class;
  }
  public boolean isCellEditable(int row, int column) {
    return column==1;
  }
  public void setValueAt(Object aValue, int row, int column) {
    employees.get(row).setManager((Boolean)aValue);
  }
  public void add(Employee employee) {
    int index=employees.size();
    employees.add(employee);
    fireTableRowsInserted(index,index);
  }
  private final ArrayList<Employee> employees=
    new ArrayList<Employee>();
}
```

```
public interface Employee {
  String getName();
  boolean isManager();
  void setManager(boolean manager);
}
```

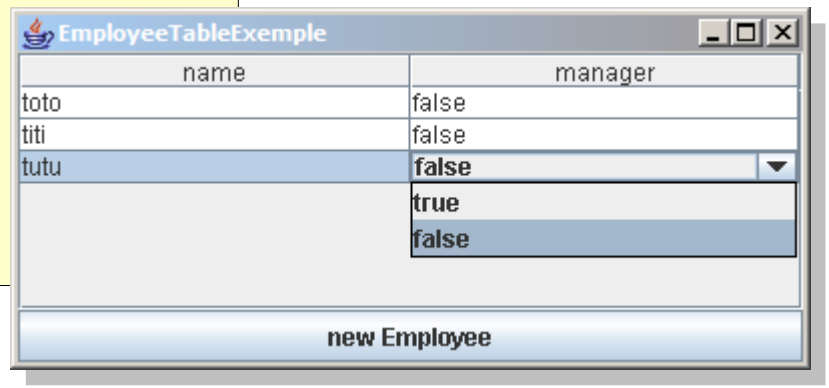


Renderer par défaut des booléens

Utilisation d'un editor à base de boite déroulante

- Il est possible de spécifier un renderer/éditeur particulier

```
final EmployeeTableModel model=new EmployeeTableModel();
JTable table=new JTable(model);

JComboBox combo=new JComboBox(new Boolean[]{true,false});
TableColumn column1=table.getColumnModel().getColumn(1);
column1.setCellRenderer(new DefaultTableCellRenderer());
column1.setCellEditor(new DefaultCellEditor(combo));

final JFrame frame=new JFrame("EmployeeTableExemple");
JButton newButton=new JButton("new Employee");
newButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    String name=JOptionPane.showInputDialog(
      frame,"employee's name");
    if (name!=null)
      model.add(createEmployee(name));
  }
});
```



Evènements de données

- Pour les données des cellules :
  TableModel obtenu par table.getModel()

```
addTableModelListener(TableModelListener l)
```

- Pour les colonnes :
  TableColumnModel obtenu par table.getColumnModel()

```
addColumnModelListener(TableColumnModelListener l)
```

Evènements de données des cellules

- Capter par l'interface **`TableModelListener`**.
- Une seule méthode :

```
void tableChanged(TableModelEvent event)
```

- L'objet **`TableModelEvent`**est paramétré différemment pour indiquer précisément le changement
  - changement de la valeur d'une ou plusieurs cellules.
  - ajout/suppression de lignes.
  - changement du nombre de colonnes.

Modification de valeurs des cellules

- Une ligne a changé de valeur :

```
TableModelEvent(source, ligne)
```

- Intervalle de ligne [ligne1,ligne2] a changé de valeur :

```
TableModelEvent(source, ligne1, ligne2)
```

- Une cellule (ligne,colonne) a changé de valeur :

```
TableModelEvent(source, ligne, ligne, colonne)
```

Ajout/Suppression de ligne

- Une colonne **ALL_COLUMNS** indique toutes les colonnes un type de modification (**INSERT,DELETE,UPDATE**)

```
TableModelEvent(source,ligne1,ligne2,ALL_COLUMNS,type)
```

- Insertion de plusieurs lignes [ligne1,ligne2] :

```
TableModelEvent(source,ligne1,ligne2,ALL_COLUMNS,INSERT)
```

- Suppression de plusieurs lignes :

```
TableModelEvent(source,ligne1,ligne2,ALL_COLUMNS,DELETE)
```

- Changement de valeur sur plusieurs lignes :

```
TableModelEvent(source,ligne1,ligne2,ALL_COLUMNS,UPDATE)
```

Modification complexe du modèle

- Le nombre de lignes et les valeurs ont changé :

```
TableModelEvent(source)
```

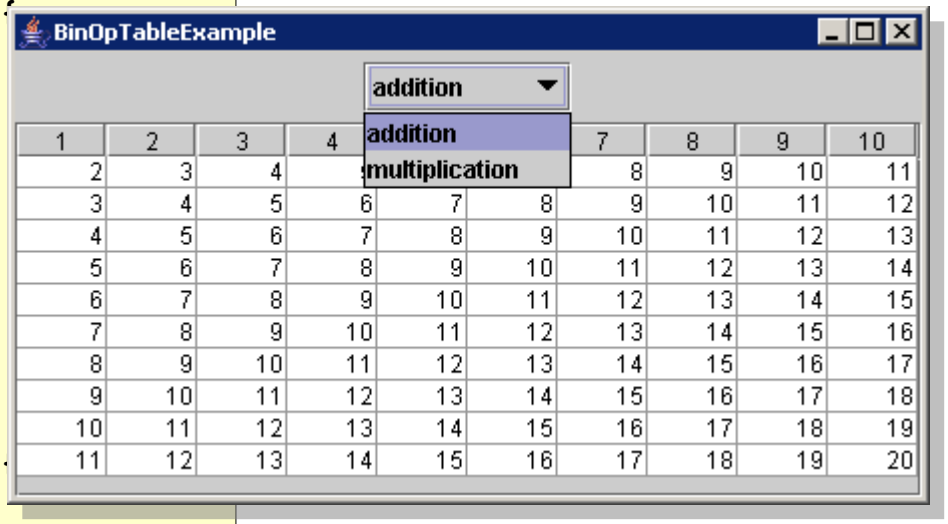
- Le nombre de colonne a changé :

```
TableModelEvent(source,HEADER_ROW)
```

- Dans ce cas le **`TableColumnModel`**est entièrement reconstruit.

Exemple de raffraichissement des données

```
class BinOpTableModel extends AbstractTableModel {
  public BinOpTableModel(BinOp binOp) {
    this.binOp=binOp;
  }
  public int getColumnCount() {
    return 10;
  }
  public int getRowCount() {
    return 10;
  }
  public Integer getValueAt(int row, int column) {
    return binOp.eval(row+1,column+1);
  }
  public String getColumnName(int column) {
    return Integer.toString(column+1);
  }
  public Class<Integer> getColumnClass(int columnIndex) {
    return Integer.class;
  }
  public void setBinOp(BinOp binOp) {
    this.binOp=binOp;
    fireTableDataChanged();
  }
  private BinOp binOp;
}
```

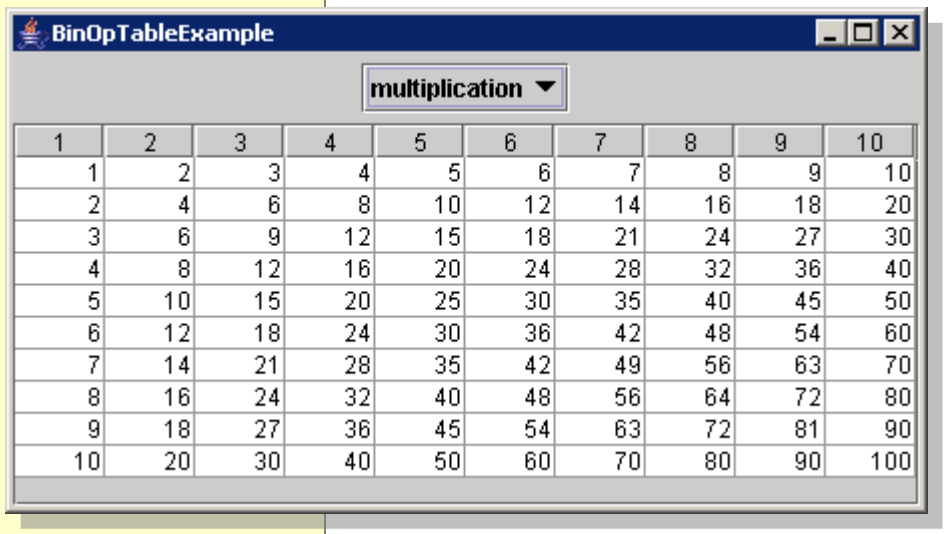


- Les valeurs dépendent d'une opération.

```
interface BinOp {
  public int eval(int left,int right);
}
```

Exemple de raffraichissement des données (2)

```
enum BinOps implements BinOp {
  addition {
    public int eval(int left, int right) {
        return left+right;
    }
  }, multiplication {
    public int eval(int left, int right) {
        return left*right;
      }
  }
}

static void main(String[] args) {
  BinOp[] ops=BinOps.values();
  final BinOpTableModel model=new BinOpTableModel(ops[0]);
  final JComboBox combo=new JComboBox(ops);
  combo.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      model.setBinOp((BinOp)combo.getSelectedItem());
    }
  });
```



1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Evènements sur les colonnes

- Capter par l'interface **`ColumnModelListener`**.
- Gestion des colonnes :
  - Ajout d'une colonne :
    `columnAdded(TableColumnModelEvent e)`
  - Supression d'une colonne :
    `columnRemoved(TableColumnModelEvent e)`
  - Déplacement d'une colonne :
    `columnMoved(TableColumnModelEvent e)`
- Changement graphique :
  - columnMarginChanged(ChangeEvent e)
  - columnSelectionChanged(ListSelectionEvent e)
- L'objet **`ColumnModelEvent`**contient deux indices indiquant la colonne de départ et celle d'arrivée.

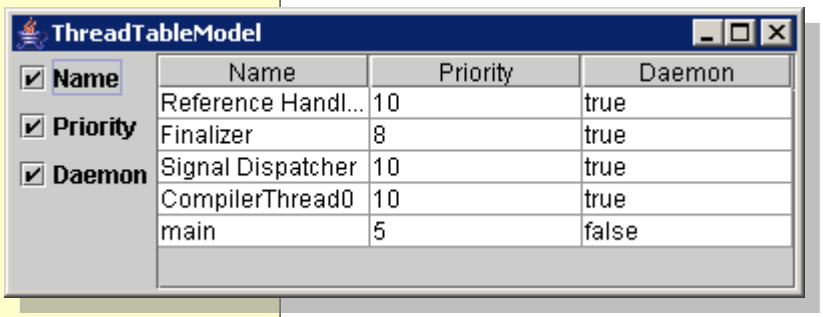
Opération sur le ColumnModel

- Les opérations de modification de colonne sur le **`ColumnModel`** génèrent automatiquement les évènements de modification correspondant.

- Opérations :
  - Obtenir les colonnes :
    `int getColumnCount()`
    `TableColumn getColumn(int columnViewIndex)`
  - ajout/suppression :
    `addColumn(TableColumn column)`
    `removeColumn(TableColumn column)`
  - Déplacement :
    `moveColumn(int columnViewIndex, int newIndex)`
  - Changement des espacements :
    `get/setColumnMargin(int margin)`

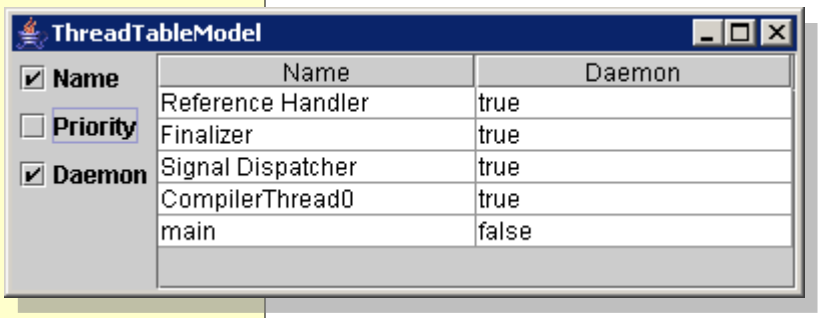
Exemple d'ajout/suppression

```
public class ThreadTableModel extends AbstractTableModel {
  public ThreadTableModel(Thread[] threads) {
    this.threads=threads;
  }
  public int getColumnCount() { return 3; }
  public int getRowCount() { return threads.length; }
  public String getColumnName(int column) {
    return columnNames[column];
  }
  public Object getValueAt(int row, int column) {
    Thread thread=threads[row];
    switch(column) {
      case 0:
        return thread.getName();
      case 1:
        return thread.getPriority();
      case 2:
        return thread.isDaemon();
    }
    throw new AssertionError(
      "invalid column ("+row+','+column+')');
  }
  private final Thread[] threads;
  private final static String[] columnNames={
    "Name","Priority","Daemon"
  };
```



Exemple d'ajout/suppression (2)

```
private static JCheckBox createCheckBox(
  final TableColumnModel columnModel,final int modelIndex) {

    final JCheckBox checkBox=new JCheckBox(
      columnNames[modelIndex]);
    checkBox.setSelected(true);
    final TableColumn column=columnModel.getColumn(modelIndex);
    checkBox.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        if (checkBox.isSelected())
          columnModel.addColumn(column);
        else
          columnModel.removeColumn(column);
      }
    });
    return checkBox;
  }
  public static JPanel createCheckBoxPanel(
    TableColumnModel columnModel) {

    JPanel panel=new JPanel(null);
    panel.setLayout(new BoxLayout(panel,BoxLayout.Y_AXIS));
    for(int i=0;i<columnModel.getColumnCount();i++)
      panel.add(createCheckBox(columnModel,i));
    return panel;
  }
```



Evènement de sélection

- Evènement de sélection différent pour les lignes et pour les colonnes.

- Evènements de sélection
  - Sélection d'une ligne (sur la JTable) : JTable.getListSelectionModel()

```
addListSelectionListener(ListSelectionListener l)
```

  - Sélection d'une colonne (sur un TableColumn) : JTable.getColumnModel().getSelectionModel()

```
addListSelectionListener(ListSelectionListener l)
```

Evènement de sélection (2)

- Par défaut seules les lignes sont sélectionnables.
  - Sélection des lignes :
    `table.setRowSelectionAllowed(true);`
  - Sélection des colonnes :
    `table.setColumnSelectionAllowed(true);`
  - Sélection des cellules :
    `table.setCellSelectionEnabled(true);`

- Méthodes de récupération de la sélection :
  - **`int getSelectedColumn()`**
  - **`int getSelectedColumnCount()`**
  - **`int[] getSelectedColumns()`**
  - **`int getSelectedRow()`**
  - **`int getSelectedRowCount()`**
  - **`int[] getSelectedRows()`**

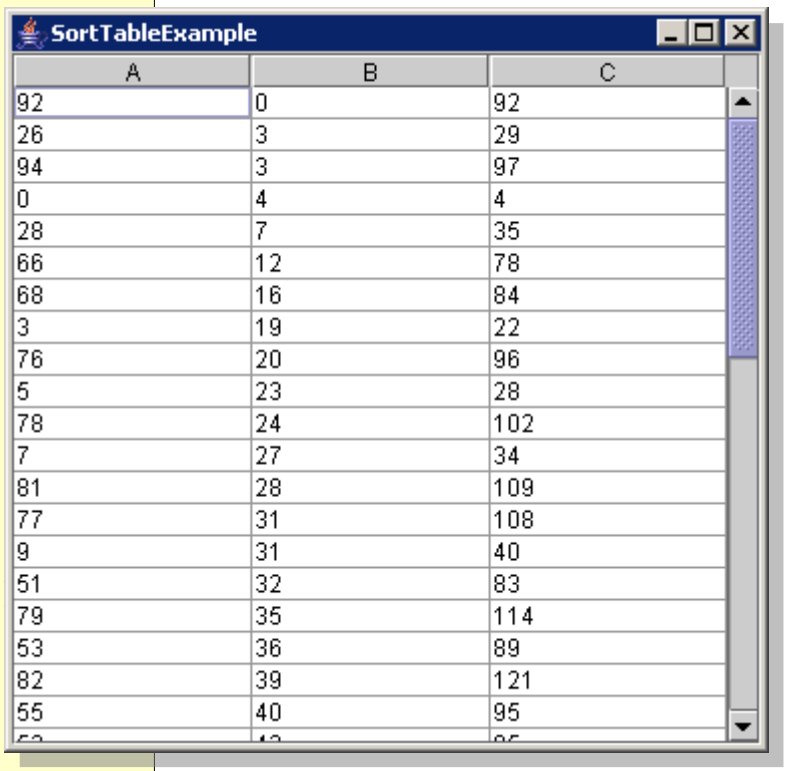
Permettre le trie suivant les colonnes

- Pas de méthode disponible dans JTable, doit être écrit à la main.
- Question : Comment permettre le trie suivant les colonnes de façon assez générique ?
- Réponse : Insérer un modèle de données entre le modèle réel et la table.
- Ce modèle effectuera des indirections sur les indices des lignes en fonction d'un trie effectué sur les valeurs d'une colonne.

Changement de l'index des lignes

- Utilisation d'un tableau d'entiers pour l'indirection.

```
public class SortedTableModel extends AbstractTableModel {
  public SortedTableModel(TableModel model) {
    this.model=model;
    Integer[] table=new Integer[model.getRowCount()];
    for(int i=0;i<table.length;i++)
      table[i]=i; // boxing
    this.table=table;
  }
  public int getColumnCount() {
    return model.getColumnCount();
  }
  public int getRowCount() {
    return model.getRowCount();
  }
  public Object getValueAt(int row, int column) {
    return model.getValueAt(getSortedRow(row),column
  }
  private int getSortedRow(int row) {
    return table[row]; // unboxing
  }
  private final Integer[] table;
  final TableModel model;
  ...
}
```



Le trie dépend des valeurs de la colonne

- Trie suivant les valeurs d'une colonne.

```
public Class<?> getColumnClass(int column) {
  return model.getColumnClass(column);
}
public String getColumnName(int column) {
  return model.getColumnName(column);
}
public boolean isCellEditable(int row, int column) {
  return model.isCellEditable(getSortedRow(row), column);
}
public void setValueAt(Object aValue, int row, int column) {
  model.setValueAt(aValue,getSortedRow(row),column);
}
public void sort(final int column,final final order) {
  Arrays.sort(table,new Comparator<Integer>() {
    public int compare(Integer i1, Integer i2) {
      Comparable c1=(Comparable)model.getValueAt(i1,column);
      Comparable c2=(Comparable)model.getValueAt(i2,column);
      return (order?1:-1)*c1.compareTo(c2); // warning unsafe
    }
  });
  fireTableDataChanged();
}
```

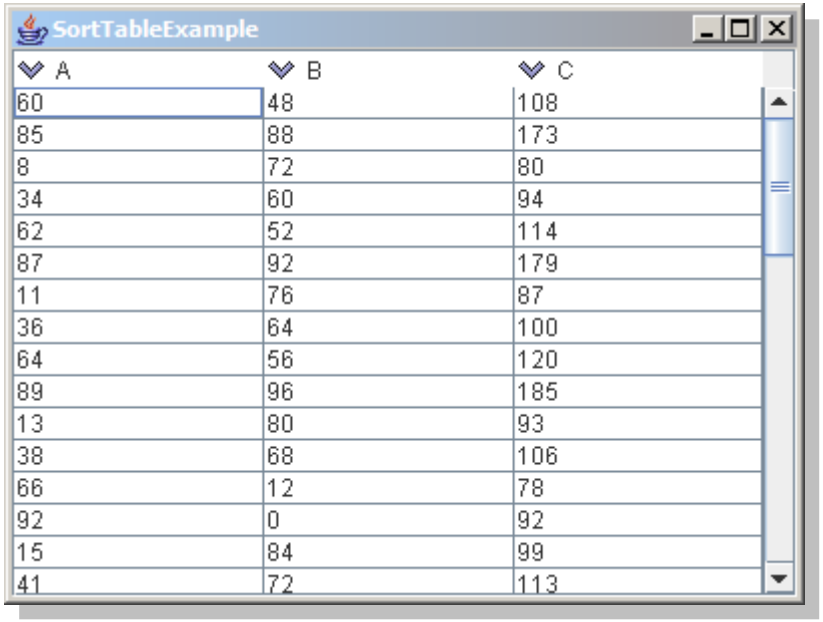
Déclenchement du tri par l'utilisateur

- L'utilisateur double-clic sur une colonne.

```
public static void main(String[] args) {
  TableModel model=new AbstractTableModel() {
    public int getColumnCount() { return 3; }
    public int getRowCount() { return 50; }
    public Integer getValueAt(int row, int column) {
      Random random=new Random(row);
      int v1=random.nextInt(100);
      int v2=random.nextInt(100);
      return new int[] {v1,v2,v1+v2}[column]; // bof
    }
  };
  final SortedTableModel sortedModel=new SortedTableModel(model);
  final JTable table=new JTable(sortedModel);
  table.getTableHeader().addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent event) {
      if (event.getClickCount()<2)
        return;
      int viewIndex = table.columnAtPoint(event.getPoint());
      int modelIndex = table.convertColumnIndexToModel(viewIndex);
      sortedModel.sort(modelIndex,true);
    }
  });
```

Trier dans les deux sens

- On souhaite maintenant pourvoir trier en ordre croissant ou décroissant
- De plus, il faut rajouter de petites flèches pour que l'utilisateur soit au courant de l'ordre de trie
- Que doit-on faire ?
  1) changer la JTable
  2) changer le TableModel
  3) changer le TableColumnModel
  4) changer le JTableHeader



- Des icônes par défaut sont disponibles : http://java.sun.com/developer/techDocs/hi/repository/

Trier dans les deux sens

```
public class SortedTableColumnModel extends DefaultTableColumnModel {
  public SortedTableColumnModel(SortedTableModel model) {
    this.model=model;
    this.orders=new boolean[model.getColumnCount()];
  }
  public boolean order(int column) {
    return orders[column];
  }
  public void reverseOrder(int column) {
    boolean order=orders[column]=!orders[column];
    model.sort(column,order);
    fireColumnMarginChanged(); // bof
  }
  private final boolean[] orders;
  private final SortedTableModel model;
}
```

```
final JTable table=new JTable(sortedModel,columnModel);
table.createDefaultColumnsFromModel();
table.getTableHeader().setDefaultRenderer(new DefaultTableCellRenderer() {
  public Component getTableCellRendererComponent(JTable table,
    Object value, boolean isSelected, boolean hasFocus, int row,int column) {
    super.getTableCellRendererComponent(table, value, isSelected,
      hasFocus, row, column);
    setIcon(columnModel.order(column)?upIcon:downIcon);
    return this;
  }
});
```