

JTABLE SWING

PHẦN 1

Nếu để ý chúng ta hẳn sẽ thấy rằng rất nhiều ứng dụng cần hiển thị dữ liệu dưới dạng bảng biểu. Trong java, Swing cung cấp cho chúng ta một lớp để thực hiện điều này đó là lớp **JTable**.

Bên cạnh khả năng hiển thị thông tin, JTable cũng cho phép chúng ta dễ dàng sửa đổi thông tin, đặt kích cỡ và đầu đề cho từng cột, và điều khiển cách dữ liệu hiển thị trong bảng biểu. Đầu tiên chúng ta hãy học cách để hiển thị thông tin lên bảng biểu vì đây là chức năng cơ bản và cũng cần thiết nhất. Bản chất của JTable là nó lấy dữ liệu từ một data model và hiển thị dữ liệu từ đó lên. Vì thế, trước tiên chúng ta sẽ nghiên cứu về cái data model.

Data model

Ngoài lớp JTable, Swing còn cung cấp một số các lớp, các interface khác nữa. Các lớp, interface này sẽ được sử dụng bởi lớp JTable và chúng được định nghĩa trong gói **javax.swing.table**. Một trong số đó là interface **TableModel**. Interface TableModel tạo nên một sự giao tiếp giữa một cái JTable và cái data model của nó. Giống như các thành phần khác của Swing, JTable được thiết kế theo mô hình model/view/controller mà ở đó các thành phần dùng để hiển thị (JTable) sẽ được tách riêng biệt so với các thành phần lưu trữ dữ liệu (TableModel). Điều này mang đến một sự linh động hơn trong thiết kế và tăng khả năng sử dụng lại các thành phần. Tuy nhiên, nó cũng làm cho việc sử dụng JTable trở nên phức tạp hơn. Nhưng cũng thật may mắn, Swing cũng cung cấp một số cài đặt mặc định làm cho việc sử dụng JTable trở nên đỡ phức tạp hơn trước.

Như đã nói, TableModel có nhiệm vụ là cung cấp dữ liệu hiển thị cho JTable. Bên cạnh đó, nó cũng có nhiệm vụ cung cấp cho JTable một vài thông tin khác như:

- Số lượng dòng và số lượng cột trong bảng
- Kiểu dữ liệu trong từng cột
- Tiêu đề cho từng cột
- Có cho phép sửa giá trị trong một ô hay không

Interface TableModel bao gồm 9 phương thức cần được cài đặt giống như dưới đây.

TableModel
<ul style="list-style-type: none"> ◆ <code>getRowCount() : int</code> ◆ <code>getColumnCount() : int</code> ◆ <code>getValueAt(row : int, column : int) : Object</code> ◆ <code>getColumnName(column : int) : String</code> ◆ <code>getColumnClass(column : int) : Class</code> ◆ <code>isCellEditable(row : int, column : int) : boolean</code> ◆ <code>addTableModelListener(listener : TableModelListener) : void</code> ◆ <code>removeTableModelListener(listener : TableModelListener) : void</code>

Cần phải cài đặt cả 9 phương thức thực sự không thích hợp trong những trường hợp chúng ta muốn tạo nhanh một bảng. Nhưng java cũng đã cung cấp cho chúng ta các lớp khác như là **AbstractTableModel** và **DefaultTableModel**. Cả hai lớp này đều đã cài đặt phần nào các phương thức của interface TableModel. Vì thế, khi sử dụng thì chúng ta sẽ tốn ít công hơn. Chẳng hạn như nếu chúng ta kế thừa lớp AbstractTableModel thì chúng ta chỉ cần cài đặt 3 phương thức sau:

- Phương thức trả về số dòng của bảng
- Phương thức trả về số cột của bảng
- Phương thức trả về giá trị của mỗi ô

Dưới đây, chúng ta tạo một lớp là **TableValues** trong file **TableValues.java**. Lớp này kế thừa lớp AbstractTableModel và sẽ là data model cho một cái JTable.

```
public class TableValues extends AbstractTableModel{

    public final static boolean GENDER_MALE = true;
    public final static boolean GENDER_FEMALE = false;

    public Object[][] values = {
        {
            "Clay", "Ashworth",
            new GregorianCalendar(1962, Calendar.FEBRUARY, 20).getTime(),
            new Float(12345.67), new Boolean(GENDER_MALE)
        }, {
            "Jacob", "Ashworth",
            new GregorianCalendar(1987, Calendar.JANUARY, 6).getTime(),
```

```

        new Float(23456.78), new Boolean(GENDER_MALE)
    }, {
        "Jordan", "Ashworth",
        new GregorianCalendar(1989, Calendar.AUGUST, 31).getTime(),
        new Float(34567.89), new Boolean(GENDER_FEMALE)
    }, {
        "Evelyn", "Kirk",
        new GregorianCalendar(1945, Calendar.JANUARY, 16).getTime(),
        new Float(-456.70), new Boolean(GENDER_FEMALE)
    }, {
        "Belle", "Spyres",
        new GregorianCalendar(1907, Calendar.AUGUST, 2).getTime(),
        new Float(567.00), new Boolean(GENDER_FEMALE)
    }
};

public int getRowCount() {
    return values.length;
}

public int getColumnCount() {
    return values[0].length;
}

public Object getValueAt(int rowIndex, int columnIndex) {
    return values[rowIndex][columnIndex];
}
}

```

Ở đây, dữ liệu chính là mảng hai chiều **values**. Vì để cho ví dụ đơn giản nên chúng ta viết dữ liệu ở trong code (hard-code). Tuy nhiên, ở ngoài thực tế, dữ liệu của chúng ta thường

sẽ lấy ra từ cơ sở dữ liệu. Như đã nói ở trên, vì kế thừa lớp `AbstractTableModel`, nên chúng ta phải cài đặt 3 phương thức đó là **`getRowCount()`**, **`getColumnCount()`** và **`getValueAt()`**. Bây giờ chúng ta tạo một lớp **`SimpleTableTest`** để tạo một `JTable` và gán cho nó cái model vừa tạo ở trên.

Đoạn code trong file **`SimpleTableTest.java`** sẽ như sau:

```
public class SimpleTableTest extends JFrame{
    protected JTable table;

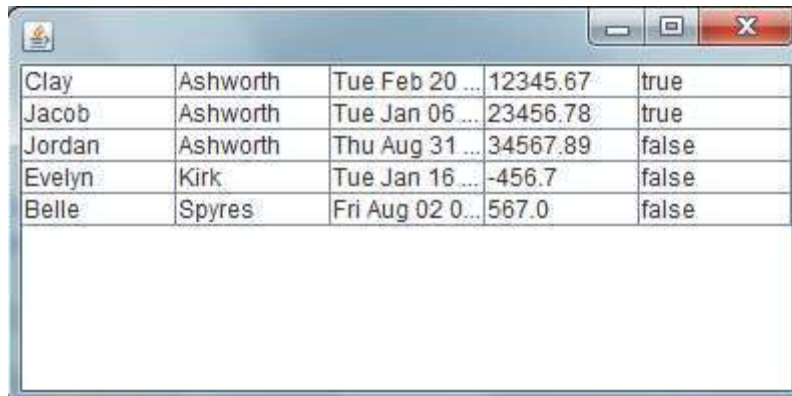
    public SimpleTableTest(){
        Container pane = getContentPane();
        pane.setLayout(new BorderLayout());
        TableValues tv = new TableValues();
        table = new JTable(tv);
        pane.add(table, BorderLayout.CENTER);
    }

    public static void main(String [] args){
        SimpleTableTest stt = new SimpleTableTest();
        stt.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        stt.setSize(400, 200);
        stt.setVisible(true);
    }
}
```

Trong constructor của lớp `SimpleTableTest` ta có hai dòng lệnh sau:

```
TableValues tv = new TableValues();
table = new JTable(tv);
```

Dòng lệnh thứ nhất là tạo ra một cái data model tên là **`tv`**. Dòng lệnh thứ hai tạo ra một cái `JTable` có tên là **`table`** và gán cho nó cái model **`tv`**. Sau khi chạy chương trình, dữ liệu trong model **`tv`** sẽ được hiển thị lên cái bảng **`table`** như sau:



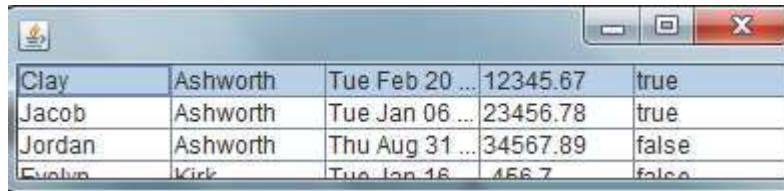
Clay	Ashworth	Tue Feb 20 ...	12345.67	true
Jacob	Ashworth	Tue Jan 06 ...	23456.78	true
Jordan	Ashworth	Thu Aug 31 ...	34567.89	false
Evelyn	Kirk	Tue Jan 16 ...	-456.7	false
Belle	Spyres	Fri Aug 02 0...	567.0	false

Như vậy, chúng ta cũng thấy được, việc sử dụng `AbstractTableModel` có vẻ nhẹ nhàng hơn rất nhiều so với việc sử dụng `TableModel`. Bên cạnh đó, việc sử dụng `DefaultTableModel` còn dễ dàng hơn nhiều so với việc sử dụng `AbstractTableModel` tuy nhiên nó lại không được khuyến khích để sử dụng. Nguyên nhân là bởi vì bản thân `DefaultTableModel` tự động tạo ra các tham chiếu đến các ô dữ liệu thay vì chúng ta phải cài đặt một phương thức nào đó giống như `getValueAt()` khi sử dụng `AbstractTableModel`. Điều này dẫn tới một sự không mềm dẻo và không thể mở rộng trong quá trình sử dụng. Bên cạnh đó, một vấn đề khác gặp phải chính là vấn đề sửa dữ liệu ở các ô. Để hiểu tại sao `DefaultTableModel` không có khả năng mở rộng, chúng ta cần phải hiểu cái `JTable` nó làm việc như thế nào.

Như chúng ta đã biết, `TableModel` có nhiệm vụ chỉ ra trong bảng có bao nhiêu dòng và có bao nhiêu cột. Hai phương thức `getRowCount()` và `getColumnCount()` sẽ được gọi ngay khi một bảng được tạo và hiển thị. Tuy nhiên, một bảng sẽ không bao giờ tham chiếu đến toàn bộ dữ liệu trong `TableModel` mà chỉ tham chiếu đến một phần nào đó để đủ hiển thị. Cho ví dụ như sau, giả sử chúng ta tạo một data model để trả về giá trị là 100 từ phương thức `getRowCount()`, nhưng cái bảng của chúng ta thì nằm trong một cái cửa sổ cuộn mà mỗi lần chúng ta chỉ nhìn tối đa được 10 dòng. Khi bảng được hiển thị, đầu tiên nó sẽ truy cập đến 10 dòng đầu của dữ liệu trong `TableModel`, và sẽ truy cập đến dữ liệu cho các dòng khác chỉ khi chúng ta cuộn thanh cuộn xuống. Tại sao điều này lại quan trọng như vậy? Bởi vì như thế nó cho phép chúng ta hiển thị một lượng lớn dữ liệu vào trong `JTable` mà không cần tải toàn bộ dữ liệu vào bộ nhớ một cách đồng thời. Lúc này, `TableModel` chỉ cần tải dữ liệu theo nhu cầu, và giảm thiểu tối đa dung lượng bộ nhớ được sử dụng.

Với điều này, giờ chúng ta quay lại việc sử dụng `DefaultTableModel` và việc nó tự tạo ra các tham chiếu đến dữ liệu bên trong nó. Bởi vì nó yêu cầu tham chiếu tới toàn bộ dữ liệu, nên toàn bộ dữ liệu phải nằm ở bên trong bộ nhớ chừng nào chúng ta vẫn còn sử dụng cái

model đó. Đây chính là nhược điểm của việc sử dụng DefaultTableModel và một lời khuyên là chúng ta nên dùng AbstractTableModel để thay thế. Tuy nhiên, với một lượng dữ liệu nhỏ, chúng ta vẫn có thể cân nhắc sử dụng DefaultTableModel bởi vì nó đơn giản. Hơn nữa dữ liệu được cache sẵn vào trong bộ nhớ sẽ giúp chúng ta truy cập nhanh hơn. Trở lại với ví dụ ở trên, chúng ta thấy rằng có một vấn đề đó là khi cửa sổ của chúng ta thay đổi sao cho kích thước của nó nhỏ hơn kích thước của bảng thì có một phần dữ liệu sẽ bị mất không thể nhìn thấy như hình dưới đây:



Clay	Ashworth	Tue Feb 20 ...	12345.67	true
Jacob	Ashworth	Tue Jan 06 ...	23456.78	true
Jordan	Ashworth	Thu Aug 31 ...	34567.89	false
Evelyn	Kirk	Tue Jan 16 ...	4567.89	false

Thêm vào nữa, các cột của chúng ta chưa có tiêu đề. Tất cả những vấn đề này sẽ được giải quyết trong phần 2.

PHẦN 2

Trong **PHẦN 1** chúng ta đã biết cách đưa dữ liệu từ một TableModel lên một JTable. Trong phần này chúng ta sẽ tiếp tục tìm hiểu về cách sử dụng JTable của Swing trong java như là tạo cửa sổ cuộn cho bảng, tạo tiêu đề cho các cột trong bảng và thao tác với hàng và cột.

Tạo cửa sổ cuộn cho bảng

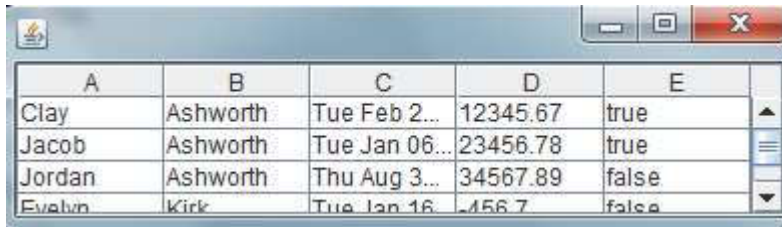
Trong bất cứ trường hợp nào mà ở đó chúng ta có một số lượng lớn thông tin cần hiển thị, chúng ta sẽ cần đến lớp **JScrollPane**. Lớp này tạo ra một cửa sổ cuộn cho phép chúng ta nhìn một lượng lớn thông tin chỉ bằng thao tác kéo thanh cuộn trên cửa sổ. Trở lại ví dụ lần trước chúng ta chỉ cần thay đổi vài dòng trong lớp SimpleTableTest trong file SimpleTableTest.java. Trong đoạn code dưới đây, những dòng in đậm màu xanh là những dòng thêm mới vào đoạn code cũ.

```
public class SimpleTableTest extends JFrame{
    protected JTable table;

    public SimpleTableTest(){
        [...]
        table = new JTable(tv);
        JScrollPane jsp = new JScrollPane(table);
        pane.add(jsp, BorderLayout.CENTER);
    }

    public static void main(String [] args){
        [...]
    }
}
```

Bây giờ chạy chương trình chúng ta sẽ thấy một thanh cuộn xuất hiện khi mà kích thước cửa sổ nhỏ hơn kích thước của bảng như hình dưới đây



A	B	C	D	E
Clay	Ashworth	Tue Feb 2...	12345.67	true
Jacob	Ashworth	Tue Jan 06...	23456.78	true
Jordan	Ashworth	Thu Aug 3...	34567.89	false
Evelyn	Kirk	Tue Jan 16...	4567	false

Nếu để ý chúng ta cũng thấy việc tạo một cửa sổ cuộn cho bảng cũng sẽ làm xuất hiện tiêu đề cho các cột. Theo mặc định, tiêu đề các cột sẽ lần lượt được đánh theo thứ tự bảng chữ cái bắt đầu từ A rồi đến B, C... Vậy để làm sao cho tiêu đề của các cột đúng với những tên mà chúng ta mong muốn. Phần dưới đây sẽ hướng dẫn tạo tiêu đề cho các cột.

Tạo tiêu đề cho các cột trong bảng

Trước tiên phải nhớ tạo ra một cửa sổ cuộn cho bảng đã. Nếu như không tạo cửa sổ cuộn cho bảng thì tiêu đề các cột cũng không thể xuất hiện. Sau đó, chúng ta có thể đặt tên cho tiêu đề của các cột trong bảng bằng cách cài đặt phương thức **getColumnName()** trong **TableModel** của chúng ta. Trong ví dụ, chúng ta chỉ việc sửa lớp **TableValues** bằng cách thêm vào các đoạn mã màu xanh sau:

```
public class TableValues extends AbstractTableModel {

    public final static boolean GENDER_MALE = true;
    public final static boolean GENDER_FEMALE = false;
    public final static String[] columnNames = {
        "First Name", "Last Name", "Date of Birth", "Account Balance", "Gender"
    };

    public Object[][] values = {
        [...]
    };

    public int getRowCount() {
        return values.length;
    }

    public int getColumnCount() {
```



```

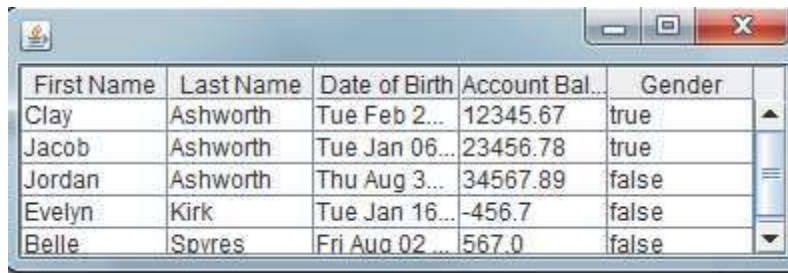
        return values[0].length;
    }

    public Object getValueAt(int rowIndex, int columnIndex) {
        return values[rowIndex][columnIndex];
    }

    @Override
    public String getColumnName(int column){
        return columnNames[column];
    }
}

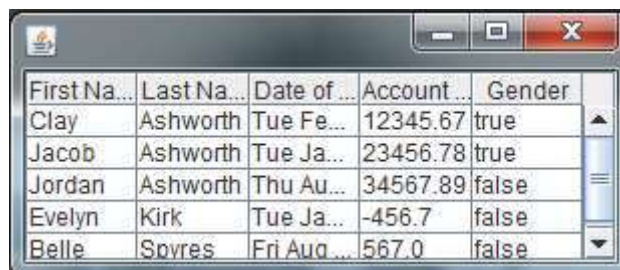
```

Giờ chạy chương trình, các tiêu đề ở các cột đã có tên rất đàng hoàng



First Name	Last Name	Date of Birth	Account Bal...	Gender
Clay	Ashworth	Tue Feb 2...	12345.67	true
Jacob	Ashworth	Tue Jan 06...	23456.78	true
Jordan	Ashworth	Thu Aug 3...	34567.89	false
Evelyn	Kirk	Tue Jan 16...	-456.7	false
Belle	Spvres	Fri Aug 02 ...	567.0	false

Chúng ta để ý rằng khi mà kích thước cửa sổ nhỏ hơn kích thước của bảng. Một thanh cuộn dọc sẽ xuất hiện nhưng lại không có một thanh cuộn ngang. Thay vào đó, các cột trong bảng sẽ tự động co lại giống như hình dưới đây.



First Na...	Last Na...	Date of ...	Account ...	Gender
Clay	Ashworth	Tue Fe...	12345.67	true
Jacob	Ashworth	Tue Ja...	23456.78	true
Jordan	Ashworth	Thu Au...	34567.89	false
Evelyn	Kirk	Tue Ja...	-456.7	false
Belle	Spvres	Fri Aug ...	567.0	false

Để hiểu tại sao điều này lại xảy ra, chúng ta cần phải tìm hiểu về thiết kế của JTable và các lớp hỗ trợ của nó làm việc như thế nào.

Thiết kế hướng cột (column-oriented design) của JTable

Thành phần JTable được thiết kế hướng cột, mỗi một JTable sẽ có một tham chiếu đến một cài đặt của interface **TableColumnModel**. Một cài đặt của TableColumnModel chẳng hạn như **DefaultTableColumnModel** nằm trong gói **javax.swing.table** và miêu tả một tập hợp các cột ở trong một cái JTable. Mỗi một cột được biểu diễn là một đối tượng của lớp **TableColumn**. Cho ví dụ như sau, giả sử chúng ta tạo một TableModel trong đó số cột của bảng được chỉ định là 5. Nếu sau đó chúng ta tạo một đối tượng của lớp JTable và đối tượng này sử dụng cái model vừa tạo thì lúc đó một đối tượng của lớp DefaultTableColumnModel sẽ được tạo ra. Đối tượng của lớp JTable sẽ lấy thông tin về số cột của bảng từ TableModel, sau đó tạo ra 5 đối tượng của lớp TableColumn rồi thêm 5 đối tượng này vào đối tượng của lớp DefaultTableColumnModel vừa tạo trước đó.

Mỗi đối tượng của lớp TableColumn mang thông tin cho một cột như tiêu đề của cột đó, giá trị về độ rộng hiện tại, độ rộng nhỏ nhất, độ rộng lớn nhất, độ rộng được ưa thích của cột đó, chỉ định xem cột đó có được phép thay đổi kích thước hay không. Ban đầu, khi mà được tạo, độ rộng hiện tại và độ rộng ưa thích của một cột được đặt giá trị là 75, độ rộng nhỏ nhất là 15 và độ rộng lớn nhất được đặt đến vô hạn (**Integer.MAX_VALUE**).

Sau khi chúng ta tạo một cột, chúng ta có thể thay đổi giá trị độ rộng của cột một cách trực tiếp bằng việc sử dụng các phương thức như **setWidth**, **setMaxWidth**, **setMinWidth**, **setMinWidth**.

Mỗi một đối tượng của lớp JTable đều được thiết lập một chế độ để chỉ ra cách nó xử lý khi mà kích thước của nó bị thay đổi. Chế độ này có thể rơi vào một trong 5 giá trị. Mỗi giá trị tương ứng với một hằng số được định nghĩa trong JTable như sau:

- **AUTO_RESIZE_ALL_COLUMNS**
- **AUTO_RESIZE_LAST_COLUMN**
- **AUTO_RESIZE_NEXT_COLUMN**
- **AUTO_RESIZE_OFF**
- **AUTO_RESIZE_SUBSEQUENT_COLUMNS**

Những giá trị này quy định cách các cột của bảng thay đổi kích thước khi mà độ rộng của bảng hoặc một trong bảng thay đổi giá trị.

Sự thay đổi kích thước của bảng

Nếu chế độ tự động thay đổi kích thước của bảng được đặt giá trị là **AUTO_RESIZE_OFF** thì việc thay đổi kích thước của bảng không làm ảnh hưởng đến kích thước hiện tại của các

cột trong bảng đó. Nhưng nếu nó nhận một trong bốn giá trị còn lại thì một sự thay đổi về kích thước của bảng sẽ được phân phối đều đến tất cả các cột trong bảng theo tỷ lệ dựa trên kích thước ưa thích (preferred width) của các cột. Cho ví dụ, giả sử rằng một bảng có hai cột trong đó một cột có độ rộng ưa thích là 200 và cột còn lại có độ rộng ưa thích là 100. Trong trường hợp này, cột thứ nhất sẽ chiếm hai phần ba chiều rộng của bảng, và cột thứ hai chiếm một phần ba độ rộng còn lại. Nếu bảng đó được kéo rộng ra thêm 30 pixel thì cột thứ nhất sẽ rộng ra thêm 20 pixels và cột thứ hai rộng ra thêm 10 pixel.

Nếu chế độ tự động thay đổi kích thước của bảng được đặt giá trị là `AUTO_RESIZE_OFF`, và tổng độ rộng của tất cả các cột lớn hơn độ rộng của bảng thì khi đó thanh cuộn ngang sẽ xuất hiện. Bây giờ quay trở lại ví dụ, ta thay đổi constructor của lớp `SimpleTableTest` như sau:

```
public class SimpleTableTest extends JFrame{
    protected JTable table;

    public SimpleTableTest(){
        [...]
        table = new JTable(tv);
        table.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
        JScrollPane jsp = new JScrollPane(table);
        pane.add(jsp, BorderLayout.CENTER);
    }

    public static void main(String [] args){
        [...]
    }
}
```

Mỗi cột có một độ rộng mặc định là 75, khi độ rộng của bảng quá nhỏ để hiển thị tất cả các cột, một thanh trượt ngang xuất hiện như hình dưới đây:



First Name	Last Name	Date of Birth	Account Bal.
Clay	Ashworth	Tue Feb 20...	12345.67
Jacob	Ashworth	Tue Jan 06...	23456.78
Jordan	Ashworth	Thu Aug 31...	34567.89
Evelyn	Kirk	Tue Jan 16...	-456.7
Belle	Spyres	Fri Aug 02 ...	567.0

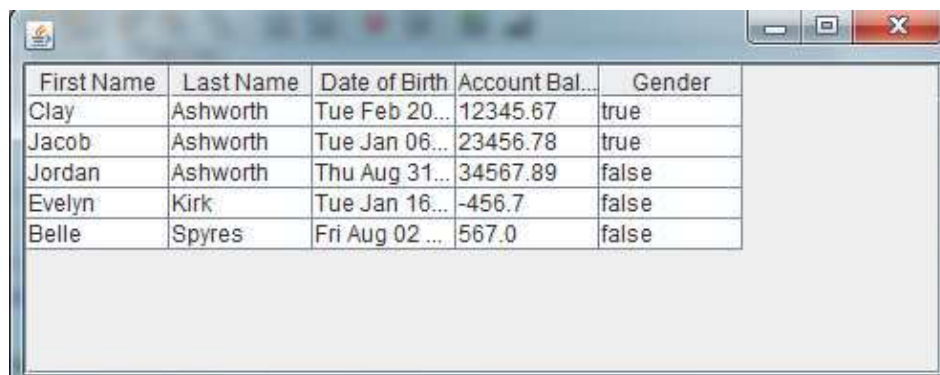
Sự thay đổi kích thước của cột

Chúng ta đã được thấy việc thay đổi độ rộng của một bảng ảnh hưởng như thế nào đến độ rộng của các cột bên trong bảng đó. Chúng ta còn phải xem xét đến việc thay đổi kích thước của một cột sẽ ảnh hưởng đến kích thước của các cột khác như thế nào. Chúng ta có thể thay đổi kích thước của cột bằng việc gọi các phương thức trong code hoặc có thể thay đổi qua giao diện của JTable.

Khi kích thước của một cột thay đổi, nó sẽ ảnh hưởng đến kích thước của các cột khác tùy thuộc vào việc chế độ tự động thay đổi kích thước của bảng được đặt giá trị như thế nào. Trong phần này, chúng ta sẽ xem xét sự thay đổi trên mỗi giá trị cụ thể.

AUTO_RESIZE_OFF

Ở chế độ này khi một cột thay đổi kích thước sẽ không làm ảnh hưởng đến kích thước của các cột khác trong bảng. Nếu kích thước của bảng nhỏ hơn tổng kích thước của tất cả các cột, thanh trượt nằm ngang sẽ xuất hiện. Nếu kích thước của bảng lớn hơn tổng kích thước của tất cả các cột, sẽ có những khoảng trống trong bảng như hình dưới đây



First Name	Last Name	Date of Birth	Account Bal...	Gender
Clay	Ashworth	Tue Feb 20...	12345.67	true
Jacob	Ashworth	Tue Jan 06...	23456.78	true
Jordan	Ashworth	Thu Aug 31...	34567.89	false
Evelyn	Kirk	Tue Jan 16...	-456.7	false
Belle	Spyres	Fri Aug 02 ...	567.0	false

AUTO_RESIZE_NEXT_COLUMN

Trong chế độ này, khi một cột thay đổi kích thước thì cột bên phải liền sát nó sẽ được nói rộng hoặc bị co lại. Trong hình dưới đây, cột Date of Birth tăng kích thước, cột phía bên phải liền sát nó là cột Account Balance sẽ bị co lại



First Name	Last Name	Date of Birth	Acco...	Gender
Clay	Ashworth	Tue Feb 20 00:00:...	123...	true
Jacob	Ashworth	Tue Jan 06 00:00:...	234...	true
Jordan	Ashworth	Thu Aug 31 00:00:...	345...	false
Evelyn	Kirk	Tue Jan 16 00:00:...	-456.7	false
Belle	Spyres	Fri Aug 02 00:00:0...	567.0	false

AUTO_RESIZE_SUBSEQUENT_COLUMNS

Trường hợp này gần giống như trường hợp sử dụng AUTO_RESIZE_NEXT_COLUMN, ngoại trừ rằng khi một cột thay đổi kích thước thì tất cả các cột phía bên phải nó sẽ được mở rộng hay co lại. Trong hình dưới đây, cột Date of Birth mở rộng ra khiến hai cột phía bên phải nó là Account Balance và Gender bị co lại



First Name	Last Name	Date of Birth	Acco...	Gen...
Clay	Ashworth	Tue Feb 20 00:00:00 ICT ...	123...	true
Jacob	Ashworth	Tue Jan 06 00:00:00 ICT ...	234...	true
Jordan	Ashworth	Thu Aug 31 00:00:00 ICT ...	345...	false
Evelyn	Kirk	Tue Jan 16 00:00:00 ICT ...	-456.7	false
Belle	Spyres	Fri Aug 02 00:00:00 ICT 1...	567.0	false

Độ chênh lệch giữa kích thước ban đầu và kích thước về sau của cột có kích thước bị thay đổi (trong ví dụ là cột Date of Birth) gọi là giá trị delta. Giá trị này được phân phối theo tỷ lệ đến các cột phía bên phải cột đó.

AUTO_RESIZE_LAST_COLUMN

Trong chế độ này, giá trị delta được phân phối đến cột cuối cùng của bảng và có thể làm cho độ rộng của nó lớn hơn hoặc nhỏ đi. Trong hình dưới đây, khi kích thước của cột Date of Birth tăng lên là nguyên nhân cột Gender bị co lại



First Name	Last Name	Date of Birth	Account Bal...	Ge...
Clay	Ashworth	Tue Feb 20 00:00:00...	12345.67	true
Jacob	Ashworth	Tue Jan 06 00:00:00...	23456.78	true
Jordan	Ashworth	Thu Aug 31 00:00:00...	34567.89	fal...
Evelyn	Kirk	Tue Jan 16 00:00:00...	-456.7	fal...
Belle	Spyres	Fri Aug 02 00:00:00 l...	567.0	fal...

AUTO_RESIZE_ALL_COLUMNS

Đây chính là chế độ mặc định cho một JTable nếu chúng ta không chỉ định chế độ tự động thay đổi kích thước cho nó. Trong chế độ này, nếu ta thay đổi kích thước cho một cột, tất cả các cột khác trong bảng cũng sẽ thay đổi kích thước theo tỷ lệ. Trong hình dưới đây, cột Date of Birth tăng kích thước làm cho tất cả các cột khác nhỏ đi.



First Na...	Last Na...	Date of Birth	Acco...	Gen...
Clay	Ashworth	Tue Feb 20 00:00:00 ICT 1962	123...	true
Jacob	Ashworth	Tue Jan 06 00:00:00 ICT 1987	234...	true
Jordan	Ashworth	Thu Aug 31 00:00:00 ICT 1989	345...	false
Evelyn	Kirk	Tue Jan 16 00:00:00 ICT 1945	-456...	false
Belle	Spyres	Fri Aug 02 00:00:00 ICT 1907	567.0	false

Đến đây, có một vấn đề là định dạng dữ liệu trong các cột chưa được theo ý muốn. Ví dụ, giá trị dữ liệu ở cột giới tính hiển thị là “true” và “false” trong khi nhẽ ra phải là “Male” và “Female” mới chuẩn. Hoặc như cột tài khoản hiển thị các giá trị kiểu số nhưng lại không theo chuẩn định dạng của tiền tệ.

Tất cả những vấn đề này sẽ được giải quyết trong phần 3.

PHẦN 3

Như đã nói ở **PHẦN 2**, ở ví dụ, dữ liệu trong các cột của bảng không hiển thị đúng như những gì mà chúng ta mong muốn. Cụ thể là ba cột sau

- Cột Date of Birth hiển thị cả ngày và giờ trong khi đó chúng ta muốn nó chỉ hiển thị ngày thôi và định dạng của ngày sinh sẽ không có thứ.
- Cột Account Balance chỉ hiển thị kiểu số đơn giản trong khi chúng ta muốn hiển thị dữ liệu ở cột này dưới định dạng của tiền tệ.
- Cột Gender hiển thị giới tính thì phải hiển thị giá trị là “Male” hoặc “Female” thay vì “true” or “false”

Định dạng biểu diễn dữ liệu cho một ô trong bảng

Mỗi một ô trong bảng JTable được vẽ nên bởi một đối tượng gọi là **cell renderer**. Đối tượng này sinh ra từ các lớp có cài đặt interface là **TableCellRenderer**. Trong interface này có định nghĩa một phương thức là **getTableCellRendererComponent()**. Phương thức này trả về một tham chiếu đến một thành phần và thành phần này sẽ thực hiện chức năng vẽ nên hình hài cho cái ô của bảng. Một cách tiện lợi nhất cho chúng ta là định nghĩa ra một lớp vừa cài đặt interface TableCellRenderer và cũng vừa kế thừa từ các lớp có khả năng thực hiện chức năng vẽ (JComboBox, JTextBox...). Chính vì vậy nên thường thường một đối tượng của lớp cài đặt interface TableCellRenderer, trong phương thức getTableCellRendererComponent() của nó sẽ trả về một tham chiếu đến chính nó. Phương thức getTableCellRendererComponent() gồm có các tham số đầu vào sau:

- Một tham chiếu đến cái JTable có các ô dữ liệu đang được vẽ
- Một tham chiếu đến giá trị của ô dữ liệu
- Một biến cờ để xác định xem ô này có đang được chọn (kích chuột vào) hay không
- Một biến cờ để xác định xem ô này có là một ô để nhập dữ liệu đầu vào và đang được trỏ đến hay không
- Một chỉ số hàng của ô đang được vẽ
- Một chỉ số cột của ô đang được vẽ

Ngoài việc trả về một tham chiếu đến một thành phần có khả năng hiển thị, phương thức getTableCellRendererComponent() còn có nhiệm vụ khởi tạo trạng thái ban đầu cho thành phần đó. Chú ý rằng một trong những tham số đầu vào của phương thức là một tham chiếu đến giá trị của một ô. Và giá trị này sẽ được lưu trong cái thành phần trước khi phương thức trả về một tham chiếu đến nó.

JTable cung cấp sẵn một số các renderers đã được định nghĩa trước, tuy nhiên trước hết chúng ta hãy tìm hiểu cách để tạo một renderer tùy biến.

Tạo renderer tùy biến

Trở lại ví dụ, chúng ta sẽ tạo một renderer tùy biến cho cột Gender, sao cho mỗi ô của nó hiển thị dưới dạng một JComboBox và giá trị “true” và “false” sẽ lần lượt được thay thế bằng “Male” và “Female”.

Trước hết ta sửa cái model TableValues trong file TableValues.java bằng cách thêm vào các hằng số, các hằng số này lần lượt lưu các chỉ số của các cột trong bảng. Cột đầu tiên sẽ có chỉ số là 0 và cứ thế tăng lên một đơn vị cho các cột tiếp sau

```
public class TableValues extends AbstractTableModel{

    public final static int FIRST_NAME = 0;
    public final static int LAST_NAME = 1;
    public final static int DATE_OF_BIRTH = 2;
    public final static int ACCOUNT_BALANCE = 3;
    public final static int GENDER = 4;

    public final static boolean GENDER_MALE = true;
    public final static boolean GENDER_FEMALE = false;

    public final static String[] columnNames = {
        "First Name", "Last Name", "Date of Birth", "Account Balance", "Gender"
    };

    public Object[][] values = {
        [...]
    };

    public int getRowCount() {
        return values.length;
    }

    public int getColumnCount() {
```



```

        return values[0].length;
    }

    public Object getValueAt(int rowIndex, int columnIndex) {
        return values[rowIndex][columnIndex];
    }

    @Override
    public String getColumnName(int column){
        return columnNames[column];
    }
}

```

Sau đó tạo một renderer có tên là **GenderRenderer**, lớp này nằm file **GenderRenderer.java** như sau:

```

public class GenderRenderer extends JComboBox implements TableCellRenderer{

    public GenderRenderer(){
        super();
        addItem("Male");
        addItem("Female");
    }

    public Component getTableCellRendererComponent(JTable table, Object value,
                                                    boolean isSelected, boolean hasFocus, int row, int column) {
        if(isSelected){
            setForeground(table.getSelectionForeground());
            super.setBackground(table.getSelectionBackground());
        }else{
            setForeground(table.getForeground());
            setBackground(table.getBackground());
        }
    }
}

```

```
        boolean isMale = ((Boolean) value).booleanValue();
        setSelectedIndex(isMale?0:1);

        return this;
    }
}
```

Như đã nói ở trên, cái renderer này sẽ cài đặt interface TableCellRenderer và để cho thuật toán nó kế thừa luôn một thành phần có khả năng hiển thị chính là JComboBox. Vì vậy, trong phương thức getTableCellRendererComponent() có trả về tham chiếu đến chính cái renderer bằng dòng lệnh **return this**.

Tiếp theo, trong lớp SimpleTableTest ở file SimpleTableTest.java, ta thay đổi như sau:

```
public class SimpleTableTest extends JFrame {
    protected JTable table;
    public SimpleTableTest() {
        [...]
        table = new JTable(tv);
        TableColumnModel tcm = table.getColumnModel();
        TableColumn tc = tcm.getColumn(TableValues.GENDER);
        tc.setCellRenderer(new GenderRenderer());
        JScrollPane jsp = new JScrollPane(table);
        pane.add(jsp, BorderLayout.CENTER);
    }
    public static void main(String [] args) {
        [...]
    }
}
```

Sau khi chạy chương trình chúng ta sẽ thấy mỗi ô trong cột Gender sẽ được biểu diễn bởi một JComboBox với giá trị là “Male” hoặc Female như hình dưới đây:

First Name	Last Name	Date of Birth	Account Bal...	Gender
Clay	Ashworth	Tue Feb 20...	12345.67	Male ▼
Jacob	Ashworth	Tue Jan 06 ...	23456.78	Male ▼
Jordan	Ashworth	Thu Aug 31...	34567.89	Female ▼
Evelyn	Kirk	Tue Jan 16 ...	-456.7	Female ▼
Belle	Spyres	Fri Aug 02 ...	567.0	Female ▼

Một điều quan trọng mà chúng ta phải biết đó là cách mà một renderer được thêm vào một đối tượng của `JTable` sẽ không giống với cách một thành phần có thể hiển thị được thêm vào một `Container`. Điều này có nghĩa rằng trong ví dụ của chúng ta cái `JTable` không thực sự chứa bất cứ đối tượng nào của `JComboBox` cả. Thay vào đó, khi cái bảng được vẽ lên, mỗi một ô sẽ có nhiệm vụ để vẽ ra những gì bên trong nó. Điều này được thực hiện bằng cách truyền một đối tượng của lớp **Graphics** vào phương thức **paint** của renderer, và khi đó vùng không gian trong một ô sẽ được vẽ lên. Với cách trên, một thành phần hiển thị có thể vẽ lên hầu hết thậm chí tất cả các ô của bảng thay vì cấp phát hẳn một thành phần hiển thị cho chỉ một ô, 100 ô sẽ cần 100 thành phần hiển thị và điều này sẽ tốn nhiều bộ nhớ của chúng ta hơn.

Trong rất nhiều trường hợp, phương pháp dễ nhất để định nghĩa một renderer là kế thừa lớp **DefaultTableCellRenderer** là renderer mặc định cho một ô trong `JTable`. `DefaultTableCellRenderer` kế thừa `JLabel`, và biểu diễn các giá trị ở các ô dưới dạng `String`. Dạng biểu diễn `String` của một đối tượng được lấy ra bằng cách gọi phương thức **toString()** của đối tượng đó, và `DefaultTableCellRenderer` sẽ truyền biểu diễn này vào phương thức **setText()** mà nó đã kế thừa từ lớp `JLabel`. Các hành động trên được thực hiện trong phương thức **setValue()**. Phương thức này nhận một tham số đầu vào là một tham chiếu đến giá trị của ô đang được vẽ.

```
public void setValue(Object value){
    setText((value == null)? "" : value.toString());
}
```

Nói tóm lại `DefaultTableCellRenderer` đơn giản là một `JLabel` với nội dung là giá trị của ô đang được vẽ.

Trong rất nhiều trường hợp, việc gọi phương thức `toString()` không phải là một phương pháp thích hợp để lấy về dạng biểu diễn String cho giá trị của một ô. Ví dụ như cột Account Balance trong ví dụ của chúng ta. Giá trị được hiển thị trong cột này thì đúng nhưng chúng không được định dạng đúng khi mà bản thân chúng đang biểu diễn các giá trị tiền tệ. Tuy nhiên, chúng ta có thể dễ dàng khắc phục vấn đề này bằng cách tạo ra một `TableCellRenderer` tùy biến và gán cho nó nhiệm vụ vẽ các giá trị dưới dạng tiền tệ. Ta đặt tên cho renderer đó là **CurrencyRenderer** và nó nằm ở trong file **CurrencyRenderer.java**

```
public class CurrencyRenderer extends DefaultTableCellRenderer{
    public CurrencyRenderer(){
        super();
        setHorizontalAlignment(javax.swing.SwingConstants.RIGHT);
    }

    @Override
    public void setValue(Object value){
        if((value != null) && (value instanceof Number)){
            Number numberValue = (Number)value;
            NumberFormat formater = NumberFormat.getCurrencyInstance();
            value = formater.format(numberValue.doubleValue());
        }
        super.setValue(value);
    }
}
```

Lớp `CurrencyRenderer` thực hiện hai việc, thứ nhất nó căn lại lề ngang cho cái `JLabel` trong suốt quá trình khởi tạo. Thứ hai là nó cài đặt lại phương thức `setValue` của `DefaultTableCellRenderer`. Bởi vì chúng ta biết rằng lớp renderer này sẽ được sử dụng để vẽ cho các ô có kiểu dữ liệu là số, cho nên chúng ta ép kiểu cho giá trị của ô về kiểu **Number** và sau đó định dạng lại giá trị này theo kiểu tiền tệ bằng việc sử dụng lớp **NumberFormat**.

Như vậy là chúng ta đã tạo một renderer tùy biến cho cột Account Balance, và chúng ta cần phải sử dụng renderer này để vẽ các ô cho cột đó. Chúng ta có thể thực hiện điều này

bằng cách trực tiếp gán cái renderer đến TableColumn giống như chúng ta đã làm trước đây đối với cột Gender. Tuy nhiên, vẫn còn một cách khác để làm điều này. Bên cạnh việc kết hợp một renderer với một cột, chúng ta còn có thể kết hợp renderer đó với một kiểu dữ liệu riêng biệt nào đó. Khi đó, trong bảng, các ô ở các cột có chung kiểu dữ liệu sẽ được vẽ chỉ bởi một renderer.

Khi JTable được khởi tạo, nó tạo một ánh xạ giữa các kiểu và các renderer. Nó sử dụng ánh xạ này để chọn ra renderer sẽ vẽ cho các ô trong các cột trong trường hợp mà các renderer cho các cột không được chỉ ra một cách tường minh. Hay nói cách khác nếu chúng ta không chỉ ra một cách trực tiếp một renderer cho một cột nào đó giống như đã làm với cột Gender thì JTable sẽ tự chọn ra renderer dựa trên kiểu dữ liệu được lưu trong cột đó. Nó xác định kiểu dữ liệu cho các cột bằng việc gọi phương thức **getColumnClass** trong TableModel. Tuy nhiên, phương thức getColumnClass() trong AbstractTableModel mặc định coi tất cả các cột của bảng sẽ có kiểu là **Object**.

```
public Class getColumnClass(int columnIndex){  
    return Object.class;  
}
```

Bởi vì AbstractTableModel mặc định không thể biết được cụ thể kiểu của các cột, cho nên chỉ có một giả thuyết an toàn đó là nó coi tất cả các ô đều có kiểu là Object. Tuy nhiên, trong thực tế, các ô sẽ hầu hết sẽ chứa các giá trị có kiểu là lớp con của lớp Object chẳng hạn như Float, Date, ... Vì vậy, nếu chúng ta muốn bảng có khả năng xác định được một kiểu riêng biệt cho một cột của nó, chúng ta phải cài đặt lại phương thức **getColumnClass()** trong TableModel của chúng ta. Cho ví dụ, bởi vì tất cả các giá trị trong cột Account Balance có kiểu là Float, nên chúng ta có thể cài đặt lại phương thức getColumnClass() trong lớp TableValues như sau:

```
public class TableValues extends AbstractTableModel{  
    public final static int FIRST_NAME = 0;  
    [...]  
  
    public Object[][] values = {  
        [...]  
    };  
}
```

```

public int getRowCount() {
    return values.length;
}

public int getColumnCount() {
    return values[0].length;
}

public Object getValueAt(int rowIndex, int columnIndex) {
    return values[rowIndex][columnIndex];
}

@Override
public String getColumnName(int column){
    return columnNames[column];
}

@Override
public Class getColumnClass(int column){
    Class dataType = super.getColumnClass(column);
    if(column == ACCOUNT_BALANCE){
        dataType = Float.class;
    }
    return dataType;
}
}

```

Bây giờ JTable đã có thể xác định được rằng cột Account Balance có kiểu là Float. Bây giờ chúng ta cần kết hợp CurrencyRenderer với kiểu dữ liệu này bằng việc gọi phương thức **setDefaultRenderer()** trong constructor của lớp SimpleTableTest

```

public class SimpleTableTest extends JFrame{
    protected JTable table;

    public SimpleTableTest(){

```

```

[...]  

tc.setCellRenderer(new GenderRenderer());  

table.setDefaultRenderer(Float.class, new CurrencyRenderer());  

JScrollPane jsp = new JScrollPane(table);  

pane.add(jsp, BorderLayout.CENTER);  

}  
  

public static void main(String [] args){  

    [...]  

}  

}

```

Với dòng lệnh màu vàng trên, CurrencyRenderer trở thành renderer mặc định cho tất cả các cột có kiểu dữ liệu là Float trong bảng. Vì vậy, CurrencyRenderer trở thành renderer cho cột Account Balance. Sau khi chạy chương trình ta thấy giá trị trong cột Account Balance có định dạng là tiền tệ giống như hình dưới đây:



First Name	Last Name	Date of Birth	Account Bal...	Gender
Clay	Ashworth	Tue Feb 20...	\$12,345.67	Male ▼
Jacob	Ashworth	Tue Jan 06 ...	\$23,456.78	Male ▼
Jordan	Ashworth	Thu Aug 31...	\$34,567.89	Female ▼
Evelyn	Kirk	Tue Jan 16 ...	(\$456.70)	Female ▼
Belle	Spyres	Fri Aug 02...	\$567.00	Female ▼

Bây giờ, điều gì sẽ xảy ra nếu như chúng ta không trực tiếp gán một renderer cho một cột và cũng không có một ánh xạ nào từ một renderer đến kiểu dữ liệu của cột đó. Khi đó, JTable sẽ thực hiện việc đi dò trên dựa trên sự kế thừa của kiểu dữ liệu trong cột bằng việc tìm kiếm một ánh xạ từ renderer đến kiểu cha của kiểu đó. Cho ví dụ, nếu phương thức getColumnClass chỉ ra rằng cột đó có kiểu là Float nhưng không có renderer nào được chỉ ra cho kiểu Float cả, khi đó JTable sẽ tìm kiếm xem có renderer nào được chỉ ra cho kiểu cha của kiểu Float là kiểu Number hay không. Nếu vẫn không có một renderer nào được chỉ ra cho kiểu Number, JTable sẽ tìm kiếm một renderer cho kiểu Object là kiểu cha của

kiểu Number. Đến đây thì nó đã tìm ra một renderer cho cột đó, chính là DefaultTableCellRenderer theo mặc định.

Vậy tổng kết lại, một JTable sẽ tìm kiếm một renderer cho các cột của nó theo các bước như sau:

1. Nếu một renderer được đặt trực tiếp cho một cột (cột Gender), sẽ sử dụng luôn renderer đó
2. Lấy về kiểu của các cột bằng việc gọi phương thức getColumnClass của TableModel
3. Nếu có một renderer nào được chỉ định đi với một kiểu, sẽ sử dụng renderer đó cho tất cả các cột có kiểu dữ liệu đó (ngoại trừ những cột đã được chỉ định trực tiếp ở trên)
4. Nếu không có renderer nào được chỉ định cho một kiểu, sẽ thực hiện tìm kiếm renderer cho kiểu cha của kiểu đó, cứ lặp lại như thế cho đến khi tìm ra một renderer thì dừng lại

Các renderer mặc định của JTable

Theo như trên, chúng ta đã biết tạo ra một renderer tùy biến của chúng ta như thế nào và làm sao để kết hợp một renderer đến một kiểu dữ liệu nào đó. Tuy nhiên, thường thì chúng ta cũng không cần để làm như thế bởi vì JTable đã cung cấp sẵn cho chúng ta một tập các renderer đã được định nghĩa trước cho một số kiểu dữ liệu thông dụng và sẽ tự động được thêm vào ánh xạ từ renderer đến kiểu dữ liệu. Một ví dụ chúng ta đã từng nói đó là có một sự kết hợp giữa kiểu Object và DefaultTableCellRenderer. Ngoài ra, còn có các renderer cho các kiểu dữ liệu khác nữa. Điều này có nghĩa rằng, nếu một trong các renderer có sẵn này có thể dùng được trong bảng của chúng ta, thì tất cả những gì chúng ta cần làm chỉ là chỉ ra kiểu dữ liệu cho các cột của bảng trong phương thức getColumnClass(). Nhờ vào đó, JTable sẽ biết cách để sử dụng các renderer có sẵn theo tương ứng với kiểu dữ liệu ở các cột. Trong ví dụ của chúng ta, chúng ta có thể sử dụng renderer có sẵn cho kiểu java.util.Date để làm renderer cho cột Date of Birth. Tất cả những gì chúng ta cần làm là thêm những dòng màu vàng sau vào phương thức getColumnClass() trong lớp TableValues như sau.

```
public class TableValues extends AbstractTableModel{  
  
    public final static int FIRST_NAME = 0;  
    [...]
```



```

public Object[][] values = {
    [...]
};

public int getRowCount() {
    return values.length;
}

public int getColumnCount() {
    return values[0].length;
}

public Object getValueAt(int rowIndex, int columnIndex) {
    return values[rowIndex][columnIndex];
}

@Override
public String getColumnName(int column){
    return columnNames[column];
}

@Override
public Class getColumnClass(int column){
    Class dataType = super.getColumnClass(column);
    if(column == ACCOUNT_BALANCE){
        dataType = Float.class;
    }else if(column == DATE_OF_BIRTH){
        dataType = java.util.Date.class;
    }
    return dataType;
}
}

```

Bây giờ chạy chương trình chúng ta sẽ thấy giá trị trong cột Date of Birth sẽ hiển thị đúng theo định dạng ngày sinh như hình dưới đây:



First Name	Last Name	Date of Birth	Account Bal...	Gender
Clay	Ashworth	Feb 20, 1962	\$12,345.67	Male
Jacob	Ashworth	Jan 6, 1987	\$23,456.78	Male
Jordan	Ashworth	Aug 31, 1989	\$34,567.89	Female
Evelyn	Kirk	Jan 16, 1945	(\$456.70)	Female
Belle	Spyres	Aug 2, 1907	\$567.00	Female

Ngoài ra còn các renderer cho các kiểu dữ liệu khác như sau:

java.lang.Number

Đây là kiểu cha cho các kiểu số như Integer, Float, Long ... Cái renderer được định nghĩa đi với kiểu Number là một lớp con của lớp DefaultTableCellRenderer và nó căn lề cho giá trị lệch sang bên phải giống như chúng ta đã làm trong CurrencyRenderer. Nói cách khác, renderer cho kiểu Number biểu diễn giá trị dưới dạng String, nhưng khác với mặc định là nó căn lề sang bên phải thay vì bên trái như mặc định. Chẳng hạn như ta áp dụng nó với cột Account Balance thì sẽ giống như hình dưới đây



First Name	Last Name	Date of Birth	Account Bal...	Gender
Clay	Ashworth	Feb 20, 1962	12,345.67	Male
Jacob	Ashworth	Jan 6, 1987	23,456.779	Male
Jordan	Ashworth	Aug 31, 1989	34,567.891	Female
Evelyn	Kirk	Jan 16, 1945	-456.7	Female
Belle	Spyres	Aug 2, 1907	567	Female

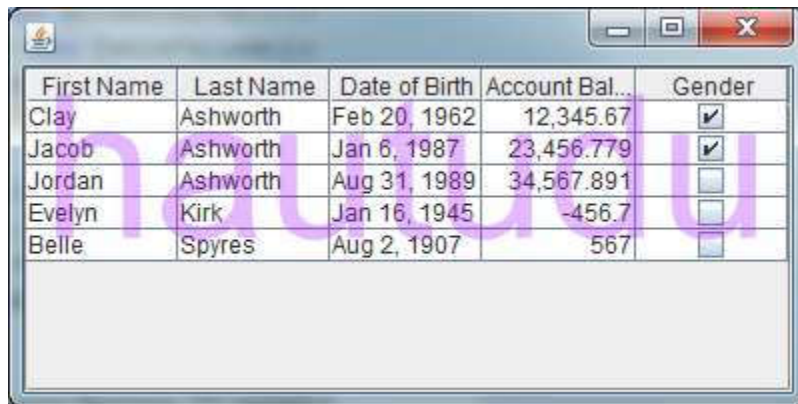
javax.swing.ImageIcon

Renderer kết hợp với lớp này cho phép chúng ta hiển thị một Icon lên trên bảng của chúng ta. Renderer này chính là DefaultTableCellRenderer và nó khai thác tính năng của một

JLabel là có thể hiển thị cả chữ và ảnh. Thay vì hiển thị chữ, nó sẽ hiển thị hình ảnh cho các ô trong cột có kiểu dữ liệu là ImageIcon.

java.lang.Boolean

Khi renderer này được sử dụng, nó sẽ hiển thị nội dung cho các ô trong các cột có kiểu là Boolean dưới dạng các JCheckBox. Các JCheckBox này sẽ ở trạng thái được đánh dấu nếu giá trị ở các ô đó là true và không được đánh dấu nếu giá trị ở các ô đó là false. Hình dưới đây minh họa điều này khi ta áp dụng nó cho cột Gender trong ví dụ.



First Name	Last Name	Date of Birth	Account Bal...	Gender
Clay	Ashworth	Feb 20, 1962	12,345.67	<input checked="" type="checkbox"/>
Jacob	Ashworth	Jan 6, 1987	23,456.779	<input checked="" type="checkbox"/>
Jordan	Ashworth	Aug 31, 1989	34,567.891	<input type="checkbox"/>
Evelyn	Kirk	Jan 16, 1945	-456.7	<input type="checkbox"/>
Belle	Spyres	Aug 2, 1907	567	<input type="checkbox"/>

Đến đây, có vẻ như chúng ta đã biết cách để hiển thị mọi thứ lên cái bảng của chúng ta. Tuy nhiên, việc sử dụng bảng không chỉ dừng lại ở mức độ hiển thị thông tin. Chúng ta còn muốn có thể trực tiếp sửa thông tin trên các ô của bảng. Điều này sẽ được nói một cách cụ thể trong phần 4.

PHẦN 4

Trong **PHẦN 3**, chúng ta đã biết cách để hiển thị dữ liệu lên trên bảng. Trong phần này, chúng ta sẽ học cách để sửa giá trị trong một ô của bảng.

Trở lại ví dụ của chúng ta mặc dù các ô trong cột Gender đều hiển thị một JComboBox, tuy nhiên chúng ta không thể thay đổi giá trị trong các ô này. Với những gì đang có, không ô nào của bảng trong ví dụ là có thể sửa được. Khi chúng ta nhấp chuột trên một ô, toàn bộ dòng đó sẽ được chọn thay vì chỉ một ô mà chúng ta muốn sửa đổi. Để thay đổi điều này, chúng ta phải cài đặt lại phương thức **isCellEditable()** trong TableModel bởi vì cài đặt mặc định của phương thức này luôn trả về giá trị là false. Chúng ta thay đổi lớp TableValues như sau:

```
public class TableValues extends AbstractTableModel{

    public final static int FIRST_NAME = 0;
    [...]

    public Object[][] values = {
        [...]
    };

    public int getRowCount() {
        return values.length;
    }

    public int getColumnCount() {
        return values[0].length;
    }

    public Object getValueAt(int rowIndex, int columnIndex) {
        return values[rowIndex][columnIndex];
    }

    @Override
    public String getColumnName(int column){
```

```


        return columnNames[column];
    }

    @Override
    public Class getColumnClass(int column){
        [...]
    }

    @Override
    public boolean isCellEditable(int row, int column){
        if(column == GENDER){
            return true;
        }
        return false;
    }
}

```

Bằng việc sửa đổi như trên, các ô trong cột Gender giờ đây đã có thể sửa được. Tuy nhiên nếu chúng ta nhấp chuột đơn vào một ô trong cột đó chúng ta lại có thêm chút ngạc nhiên bởi vì không có điều gì xảy ra ngoại trừ cả dòng đó được chọn. Trong khi đó, cái mà chúng ta mong chờ đó là có thể chọn một giá trị khác từ một JComboBox. Nếu chúng ta kích đúp chuột vào ô đó, một JTextField xuất hiện và trong nó là giá trị kiểu Boolean của ô Gender (true hoặc false), và chúng ta có thể sửa giá trị này trong JTextField giống như hình dưới đây:



First Name	Last Name	Date of Birth	Account Bal...	Gender
Clay	Ashworth	Feb 20, 1962	\$12,345.67	Male ▼
Jacob	Ashworth	Jan 6, 1987	\$23,456.78	Male ▼
Jordan	Ashworth	Aug 31, 1989	\$34,567.89	false
Evelyn	Kirk	Jan 16, 1945	(\$456.70)	Female ▼
Belle	Spyres	Aug 2, 1907	\$567.00	Female ▼

Tại sao lại là một `JTextField` mà lại không phải là một `JComboBox`. Để giải thích điều này chúng ta phải nhớ rằng các ô trong bảng không thực sự chứa bất cứ một thành phần nào. Các ô chỉ đơn giản là được vẽ bởi các renderer và trong trường hợp này các ô trong cột Gender được vẽ như là các `JComboBox`. Quá trình sửa đổi giá trị trong các ô là một quá trình hoàn toàn tách rời riêng biệt và nó có thể được xử lý bởi một thành phần giống như là thành phần được vẽ bởi renderer. Ví dụ, thành phần hiển thị mặc định cho một `JTable` là một `JLabel` trong khi thành phần sửa đổi mặc định của nó là một `JTextField` và đây là lý do tại sao `JTextField` lại xuất hiện khi mà chúng ta chọn để sửa một ô trong cột Gender. Bất kể là thành phần nào được sử dụng, mục tiêu cuối cùng vẫn là làm sao để cho giá trị trong một ô có thể thay đổi được. Tuy nhiên trong trường hợp của chúng ta giá trị sau khi được sửa đổi sẽ bị loại bỏ và lại quay trở về giá trị ban đầu. Để hiểu tại sao điều này lại xảy ra và chúng ta phải làm những gì, trước hết chúng ta hãy làm quen với khái niệm editor cho một ô và biết cách một `JTable` xử lý sự thay đổi giá trị trong các ô của nó như thế nào.

Editor cho một ô trong `JTable`

Giống như renderer điều khiển cách hiển thị giá trị của một ô, editor xử lý khi mà giá trị trong ô đó được sửa đổi. Editor có thể nói là phức tạp hơn so với renderer một chút nhưng nó so với renderer thì cũng có nhiều điểm tương đồng như sau:

- Một editor có thể gán tới một hay nhiều đối tượng của lớp `TableColumn`
- Một editor có thể gán tới một hay nhiều kiểu dữ liệu và sẽ được sử dụng để làm editor của các cột có kiểu dữ liệu đó trong trường hợp không có một editor nào được chỉ định trực tiếp để kết hợp với các cột đó.
- Các thành phần hiển thị có thể được sử dụng trong quá trình chỉnh sửa, giống như cách mà chúng được sử dụng bởi các renderer. Trong thiết kế thực tế, một renderer và một editor cho một ô của bảng thường sử dụng chung một thành phần hiển thị.

Chúng ta có thể gán editor đến một đối tượng của `TableColumn` hoặc đến một kiểu dữ liệu bằng cách lần lượt sử dụng phương thức **`setCellEditor()`** trong `TableColumn` hoặc phương thức **`setDefaultEditor()`** trong `JTable`. Việc cài đặt interface **`TableCellEditor`** thì phức tạp hơn so với trường hợp cài đặt `TableCellRenderer` và để hiểu được các phương thức trong `TableCellEditor`, chúng ta hãy tìm hiểu editor tương tác với `JTable` như thế nào.

Khi một `JTable` phát hiện ra một thao tác nhấp chuột trên một ô, nó gọi phương thức **`isCellEditable()`** trong `TableModel`. Nếu phương thức này trả về một giá trị là `false` thì khi đó có nghĩa là ô đó không thể sửa được và quá trình xử lý dừng ở đây, không làm thêm bất

cứ gì khác nữa. Trong trường hợp phương thức này trả về giá trị là true thì sau đó JTable sẽ nhận diện editor của ô này và gọi phương thức **isCellEditable()** của editor. Mặc dù TableModel và CellEditor đều có phương thức là isCellEditable(), nhưng giữa chúng có một sự khác biệt. Phương thức trong TableModel chỉ lấy giá trị của chỉ số cột và chỉ số dòng làm tham số đầu vào, trong khi đó, phương thức trong CellEditor lấy tham số đầu vào là một đối tượng của **EventObject** biểu diễn cho sự kiện kích chuột. Chúng ta có thể sử dụng đối tượng này để lấy ra số lần kích chuột trong sự kiện. Một ô phải được kích đúng trước khi chúng ta có thể sửa nó, và chúng ta đã quan sát hiện tượng này trong cột Gender của ví dụ. Nói một cách khác phương thức isCellEditable() trả về một giá trị là false khi mà số lần kích chuột là 1, và sẽ trả về là true nếu như số lần kích chuột lớn hơn 1.

Chúng ta có thể thực hiện sửa đổi giá trị cho một ô khi và chỉ khi cả hai phương thức isCellEditable() trên đều trả về giá trị là true. Khi trường hợp này xảy ra, quá trình sửa đổi được khởi tạo bằng việc gọi phương thức **getTableCellEditorComponent()**. Phương thức này gồm những tham số đầu vào sau:

- Một tham chiếu đến JTable của ô đang được sửa
- Một tham chiếu đến giá trị hiện thời của ô
- Một biến boolean để xác định xem ô đó có được chọn hay không
- Chỉ số hàng của ô đang được sửa
- Chỉ số cột của ô đang được sửa

Các tham số ở trên tương tự với các tham số đầu vào trong phương thức getTableCellRendererComponent() trong TableCellRenderer. Sự khác biệt duy nhất ở đây chính là phương thức getTableCellEditorComponent() không có một tham số đầu vào kiểu boolean để chỉ ra ô đó có phải là một ô để nhập dữ liệu và đang được trỏ đến hay không, bởi vì cái này có thể tự suy diễn ra luôn là true khi mà ô đó đang ở trạng thái được sửa.

Trước khi trả về một tham chiếu đến thành phần hiển thị cho việc xử lý chỉnh sửa, phương thức getTableCellEditorComponent() phải thực hiện khởi tạo sao cho giá trị thể hiện trên thành phần phải trùng khớp với giá trị hiện thời của ô đó. Giả sử, chúng ta đang tạo một editor để cho phép người sử dụng chọn giá trị cho cột Gender là Male hoặc Female từ một JComboBox. Trong trường hợp đó, chúng ta cần phải khởi tạo JComboBox sao cho ban đầu giá trị được chọn trong nó phải giống với giá trị hiện thời của các ô trong cột Gender. Sau khi thành phần tham chiếu đã được trả về từ phương thức getTableCellEditorComponent(), JTable sẽ đặt kích thước và vị trí cho thành phần đó sao cho nó trực tiếp nằm đè trùng khớp lên ô đang được sửa. Điều này làm cho ta tưởng rằng

ô đó đang thực sự ở trạng thái chỉnh sửa nhưng thực ra là các thao tác chỉnh sửa chỉ thực hiện trên một thành phần nằm phía trên cái ô đó.

Việc đặt thành phần dùng cho việc chỉnh sửa bên trên một ô cũng có nghĩa là sự kiện khởi tạo quá trình chỉnh sửa được truyền vào thành phần này. Ví dụ, trong trường hợp một editor dùng JComboBox cho việc chỉnh sửa thì sự kiện kích chuột được truyền vào JComboBox. Chính vì thế khi kích chuột vào một ô để chỉnh sửa cũng có nghĩa là JComboBox cũng nhận sự kiện này và nó sẽ sổ ra một cửa sổ với nhiều lựa chọn. Chúng ta chọn một trong số chúng cũng bằng thao tác kích chuột. Cuối cùng của quá trình này, khi phương thức **shouldSelectCell()** của editor được truyền vào sự kiện kích chuột trên và trả về giá trị là true thì đây là lúc việc chỉnh sửa kết thúc.

Mỗi một editor cần phải cài đặt phương thức **addCellEditorListener()** và **removeCellEditorListener()**, và interface **CellEditorListener** có định nghĩa hai phương thức là **editingStopped()** và **editingCanceled()**. Trong thực tế, listener thường chính là bản thân JTable, nó sẽ được thông báo khi mà việc chỉnh sửa một ô dừng lại hoặc hủy bỏ. Thêm vào nữa, editor còn phải cài đặt hai phương thức nữa là **cancelCellEditing()** và **stopCellEditing()**. Hai phương thức này lần lượt gọi hai phương thức **editingCanceled()** và **editingStopped()** của listener đã đăng kí.

Một yêu cầu để kết thúc việc chỉnh sửa cho một ô có thể bắt nguồn từ JTable chứa ô đó hoặc từ ngay chính bản thân editor. Lấy ví dụ, giả sử chúng ta kích chuột vào một ô và bắt đầu chỉnh sửa giá trị của nó. Nếu sau đó chúng ta kích sang một ô thứ hai khác, thì JTable sẽ gọi phương thức **stopCellEditing()** trong editor của ô thứ nhất trước khi nó khởi tạo quá trình chỉnh sửa cho ô thứ hai. Một editor hoàn toàn có thể dừng việc chỉnh sửa khi một sự kiện nào đó xảy ra để chỉ định việc chỉnh sửa đã hoàn tất. Cho ví dụ, khi sử dụng một JComboBox như một editor, nếu nó nhận một thông điệp từ **ActionEvent** thì nó ngầm định rằng đây là dấu hiệu chúng ta vừa thực hiện chọn một giá trị và kết thúc quá trình chỉnh sửa. Tương tự như vậy, một JTextField nhận một tín hiệu để dừng việc chỉnh sửa đó chính là khi phím Enter được bấm.

Bất kể là các yêu cầu cho việc dừng chỉnh sửa được sinh ra ở đâu thì phương thức **editingStopped()** của JTable vẫn cứ được gọi bởi vì nó thuộc về một **CellEditorListener** đã được đăng kí. Trong phương thức này, JTable gọi phương thức **getCellEditorValue()** của editor để lấy về giá trị mới trong ô được chỉnh sửa sau đó truyền giá trị này vào phương

thức **setValueAt()** trong **TableModel** của **JTable**. Điều này có nghĩa rằng, giá trị mới của ô được lấy về từ editor và được gửi đến model để có thể được lưu trữ.

Trở về với ví dụ, giờ ta tạo một editor đặt tên là **GenderEditor** cho cột **Gender**. Editor này sử dụng một **JComboBox** để cho phép chúng ta có thể chọn một trong hai giá trị là **Male** hoặc **Female** trong cột **Gender**. Tạo lớp **GenderEditor** trong file **GenderEditor.java** như sau

```
public class GenderEditor extends JComboBox implements TableCellEditor{
```

```
    protected EventListenerList listenerList = new EventListenerList();
```

```
    protected ChangeEvent changeEvent = new ChangeEvent(this);
```

```
    public GenderEditor(){
```

```
        super();
```

```
        addItem("Male");
```

```
        addItem("Female");
```

```
        addActionListener(new ActionListener() {
```

```
            public void actionPerformed(ActionEvent e) {
```

```
                fireEditingStopped();
```

```
            }
```

```
        });
```

```
    }
```

```
    public void addCellEditorListener(CellEditorListener listener){
```

```
        listenerList.add(CellEditorListener.class, listener);
```

```
    }
```

```
    public void removeCellEditorListener(CellEditorListener listener){
```

```
        listenerList.remove(CellEditorListener.class, listener);
```

```
    }
```

```
    protected void fireEditingStopped(){
```

```
        CellEditorListener listener;
```

```
Object[] listeners = listenerList.getListenerList();
for(int i=0; i<listeners.length; i++){
    if(listeners[i] == CellEditorListener.class){
        listener = (CellEditorListener) listeners[i+1];
        listener.editingStopped(changeEvent);
    }
}
}
```

```
protected void fireEditingCanceled(){
    CellEditorListener listener;
    Object[] listeners = listenerList.getListenerList();
    for(int i=0; i< listeners.length; i++){
        if(listeners[i] == CellEditorListener.class){
            listener = (CellEditorListener) listeners[i+1];
            listener.editingCanceled(changeEvent);
        }
    }
}
```

```
public void cancelCellEditing(){
    fireEditingCanceled();
}
```

```
public boolean stopCellEditing(){
    fireEditingStopped();
    return true;
}
```

```
public boolean isCellEditable(EventObject event){
    return true;
}
```

```

public boolean shouldSelectCell(EventObject event){
    return true;
}

public Object getCellEditorValue(){
    return new Boolean(getSelectedIndex() == 0 ? true : false);
}

public Component getTableCellEditorComponent(JTable table, Object value,
                                              boolean isSelected, int row, int column){
    boolean isMale = ((Boolean) value).booleanValue();
    setSelectedIndex(isMale?0:1);
    return this;
}
}

```

Khi việc chỉnh sửa được kích hoạt trong một ô nào đó của cột Gender, phương thức `getTableCellEditorComponent()` được gọi, editor lúc này chuẩn bị trạng thái cho nó trước khi hiển thị. Trong trường hợp này, giá trị Male hoặc Female sẽ được chọn dựa trên giá trị hiện tại của ô đang được chỉnh sửa. Khi chúng ta chọn một giá trị trong `JComboBox`, phương thức `fireEditingStopped()` được gọi, và nó thông báo với `JTable` rằng quá trình chỉnh sửa kết thúc. `JTable` sau đó sẽ gọi phương thức `getCellEditorValue()` để lấy về giá trị mới của ô đó và sẽ truyền giá trị này vào phương thức `setValueAt()` của `TableModel`. Tiếp theo chúng ta sửa lại lớp `SimpleTableTest` trong file `SimpleTableTest.java` như sau:

```

public class SimpleTableTest extends JFrame{
    protected JTable table;

    public SimpleTableTest(){
        [...]
        TableColumnModel tcm = table.getColumnModel();
        TableColumn tc = tcm.getColumn(TableValues.GENDER);
        tc.setCellRenderer(new GenderRenderer());
        tc.setCellEditor(new GenderEditor());
    }
}

```

```

        table.setDefaultRenderer(Float.class, new CurrencyRenderer());
        JScrollPane jsp = new JScrollPane(table);
        pane.add(jsp, BorderLayout.CENTER);
    }

    public static void main(String [] args){
        [...]
    }
}

```

Khi chạy chương trình ta sẽ thấy khi kích vào một ô trong cột Gender để chỉnh sửa, một cửa sổ được mở xuống với giá trị được bôi đen chính là giá trị hiện thời của ô đó.



Để cho sau khi chọn giá trị này thực sự được thay đổi hẳn thì chúng ta cần phải cài đặt lại phương thức setValueAt() trong lớp TableValues như sau

```

public class TableValues extends AbstractTableModel{

    public final static int FIRST_NAME = 0;
    [...]

    public Object[][] values = {
        [...]
    };

    public int getRowCount() {

```

```
        return values.length;
    }

    public int getColumnCount() {
        return values[0].length;
    }

    public Object getValueAt(int rowIndex, int columnIndex) {
        return values[rowIndex][columnIndex];
    }

    @Override
    public String getColumnName(int column){
        return columnNames[column];
    }

    @Override
    public Class getColumnClass(int column){
        [...]
    }

    @Override
    public boolean isCellEditable(int row, int column){
        [...]
    }

    @Override
    public void setValueAt(Object value, int row, int column){
        values[row][column] = value;
    }
}
```

DefaultCellEditor

Chúng ta cũng không nhất thiết trong mọi trường hợp đều phải tự xây dựng một editor hoàn toàn mới. Trong thực tế, lớp DefaultCellEditor cho phép chúng ta dễ dàng tạo một editor sử dụng một JCheckBox, JComboBox, hoặc là JTextField. Tất cả mọi thứ phải làm là tạo một đối tượng của lớp DefaultCellEditor và sau đó truyền cho nó các đối tượng của một trong 3 thành phần hiển thị kể trên vào. Tuy nhiên, DefaultCellEditor không có tính mềm dẻo và trong nhiều trường hợp chúng ta vẫn phải tạo editor của riêng chúng ta.

Như vậy trong phần 4, chúng ta đã biết cách để chỉnh sửa dữ liệu ngay chính trên cái bảng của chúng ta. Các thao tác chọn thực hiện trên hàng và cột trong một bảng sẽ được giới thiệu trong phần 5.

PHẦN 5

Trong phần này, chúng ta tập trung vào một số thiết lập để làm việc với các thao tác chọn trên một bảng.

JTable là một thành phần hai chiều (hàng và cột): mỗi ô được chọn sẽ có chỉ số hàng và chỉ số cột. Trong khi đó, JList là một thành phần một chiều mà ở đó mỗi một ô chỉ có duy nhất một chỉ số đó là chỉ số hàng. Bởi vì bản chất của JTable là một thành phần hai chiều cho nên thông tin được chọn của bảng không thể được quản lý bởi một **ListSelectionModel**. Interface ListSelectionModel chỉ hỗ trợ cho việc quản lý các thông tin được chọn ở hình thức một chiều. Để khắc phục vấn đề này, Jtable sử dụng hai đối tượng của lớp **DefaultListSelectionModel**.

- Một đối tượng được quản lý bởi chính bản thân Jtable. Đối tượng này quản lý thông tin được chọn theo dòng
- Một đối tượng được quản lý bởi TableColumnModel. Đối tượng này quản lý thông tin được chọn theo cột

Như chúng ta đã nói lúc trước, khi một ô được chọn, các ô khác cũng có thể được chọn theo. Điều này tùy thuộc vào việc chúng ta thiết lập cách chọn cho bảng. Trong thực tế, có rất nhiều thiết lập chọn cho một bảng. Interface ListSelectionModel và lớp cài đặt của nó là DefaultListSelectionModel được sử dụng để quản lý việc chọn trên một JTable.

- **Chọn theo hàng:** Khi một ô được chọn thì tất cả các ô khác cùng hàng với nó cũng được chọn. Đây là thiết lập chọn mặc định của JTable. JTable cung cấp cho chúng ta hai phương thức là **getRowSelectionAllowed()** và **setRowSelectionAllowed()** để cho phép chúng ta truy vấn, thiết lập, hoặc hủy bỏ cơ chế chọn theo hàng cho bảng.
- **Chọn theo cột:** Khi một ô được chọn, tất cả các ô khác cùng cột với nó cũng được chọn. Việc truy vấn, thiết lập, hủy bỏ cơ chế chọn theo cột cho bảng được thực hiện bởi hai phương thức **getColumnSelectionAllowed()** và **setColumnSelectionAllowed()** của JTable
- **Chọn theo ô:** Trong chế độ này, việc chọn một ô sẽ không ảnh hưởng gì đến các ô khác. Việc truy vấn, thiết lập, hủy bỏ cơ chế chọn theo ô cho bảng được thực hiện bởi hai phương thức **getCellSelectionEnabled()** và **setCellSelectionEnabled()** của JTable. Việc kích hoạt chế độ chọn theo ô sẽ làm vô hiệu hóa hai chế độ là chọn theo hàng và chọn theo cột

Kết hợp các chế độ chọn theo hàng, chọn theo cột và chọn theo ô

Chúng ta có thể sử dụng chế độ chọn theo hàng và chọn theo cột cùng nhau sao cho khi chúng ta nhấp chuột để chọn một ô, tất cả các ô khác trong cùng hàng hoặc cột với nó cũng được chọn. Tuy nhiên việc kích hoạt chế độ chọn theo ô sẽ dẫn đến việc các chế độ chọn theo hàng và chọn theo cột bị mất hiệu lực. Cho ví dụ, giả sử chúng ta có đoạn mã sau:

```
JTable table;  
...  
table.setRowSelectionAllowed(true);  
table.setColumnSelectionAllowed(true);  
table.setCellSelectionEnabled(true);
```

Trong đoạn mã trên, mặc dù tất cả các chế độ chọn của bảng đều được kích hoạt tuy nhiên việc kích hoạt chế độ chọn theo ô sẽ vô hiệu hóa hai chế độ kia. Vì thế, mặc dù chúng ta có ba loại thiết lập chế độ chọn cho bảng, nhưng chỉ có bốn tổ hợp của chúng là có ý nghĩa:

- Chỉ có chế độ chọn theo hàng được kích hoạt
- Chỉ có chế độ chọn theo cột được kích hoạt
- Chế độ chọn theo ô được kích hoạt. Khi đó chúng ta không cần quan tâm đến việc hai chế độ còn lại có được kích hoạt hay không
- Khi mà tất cả các chế độ chọn bị vô hiệu hóa.

Trong trường hợp cuối cùng, khi mà tất cả các chế độ bị vô hiệu hóa, chúng ta không thể chọn bất kì một ô nào trong bảng.

Các chế độ chọn theo danh sách

Khi một ô được chọn, lúc đó một hoặc cả hai đối tượng của `ListSelectionModel` sẽ được cập nhật theo việc chọn này. Theo mặc định, mỗi model có thể duy trì nhiều khoảng giá trị. Cho ví dụ, như hình dưới đây, trong model sẽ duy trì hai khoảng giá trị được chọn. Khoảng giá trị thứ nhất là các dòng trong bảng có chỉ số từ 0 đến 1 và khoảng giá trị thứ hai là các dòng trong bảng có chỉ số từ 3 đến 4.



First Name	Last Name	Date of Birth	Account Bal...	Gender
Clay	Ashworth	Feb 20, 1962	\$12,345.67	Male ▼
Jacob	Ashworth	Jan 6, 1987	\$23,456.78	Male ▼
Jordan	Ashworth	Aug 31, 1989	\$34,567.89	Female ▼
Evelyn	Kirk	Jan 16, 1945	(\$456.70)	Female ▼
Belle	Spyres	Aug 2, 1907	\$567.00	Female ▼

Để chọn ra hai khoảng giá trị giống như trên, chúng ta làm các bước như sau:

1. Chọn dòng trên cùng của bảng
2. Bấm và giữ phím Shift sau đó nhấp chuột chọn dòng thứ hai. Như vậy đến đây, khoảng thứ nhất của các dòng đã được chọn
3. Thả phím Shift sau đó bấm và giữ phím Ctrl, chọn dòng thứ tư
4. Thả phím Ctrl, bấm và giữ phím Shift sau đó nhấp chuột chọn dòng cuối cùng. Lúc này, khoảng thứ hai của các dòng được chọn

Như chúng ta đã làm phía trên, để chọn một khoảng giá trị giữa hai cặp giá trị, chúng ta chỉ việc chọn điểm đầu khoảng giá trị sau đó bấm và giữ phím Shift rồi chọn điểm cuối khoảng giá trị đó. Trong khi đó việc bấm phím Ctrl sẽ cho phép chúng ta chọn một giá trị khác nhưng không làm mất đi những giá trị hay khoảng giá trị đã được chọn trước đó. Một cách chọn khác thay vì sử dụng phím Shift để chọn một khoảng giá trị đó là nhấp chuột lên rồi kéo thả con trỏ trên khoảng giá trị đó. Cho ví dụ, trong trường hợp này, chúng ta có thể nhấp chuột lên một ô của dòng đầu tiên và sau đó kéo thả chuột đến dòng thứ hai, như vậy khoảng giá trị đầu tiên trong bảng sẽ được chọn.

Ví dụ trên đã minh họa cho chế độ chọn mặc định gọi là **multiple-interval selection** hay là có thể chọn nhiều khoảng giá trị. Đây là một trong ba chế độ mà ListSelectionModel hỗ trợ. Hai chế độ còn lại là **single-interval selection** và **single selection**.

Như cái tên đã chỉ ra, chế độ single-interval selection là chế độ cho phép chọn một khoảng giá trị đơn lẻ. Trong ví dụ của chúng ta, nếu chúng ta chọn chế độ single-interval selection thì sau khi khoảng giá trị thứ hai (từ chỉ số hàng 3 đến 4) được chọn, khoảng giá trị thứ nhất (từ chỉ số hàng 0 đến 1) được chọn trước đó sẽ trở thành không được chọn.

Còn trong chế độ single-selection, một ListSelectionModel chỉ cho phép chọn một giá trị đơn lẻ và không thể chọn một khoảng giá trị trong bảng. Cho ví dụ, khi chúng ta kích hoạt

chế độ chọn theo cột và single-selection, chúng ta chỉ có thể chọn một cột duy nhất tại một thời điểm.

Như đã nói trước đó, mỗi JTable sẽ duy trì hai đối tượng của ListSelectionModel và cung cấp phương thức **setSelectionMode()** để thiết lập chế độ chọn theo danh sách cho cả hai đối tượng đó. Mỗi một chế độ chọn sẽ được biểu diễn bởi một giá trị hằng số được định nghĩa trong ListSelectionModel:

- MULTIPLE_INTERVAL_SELECTION
- SINGLE_INTERVAL_SELECTION
- SINGLE_SELECTION

Chú ý rằng JTable không có phương thức **getSelectionMode()** để lấy về chế độ hiện tại. Phương thức này thuộc về đối tượng của ListSelectionModel. Chúng ta phải lấy thông tin trên từ đối tượng của ListSelectionModel như cách sau:

```
JTable table;  
...  
int oldSelectionMode = table.getSelectionModel().getSelectionMode();  
table.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
```

Trở lại ví dụ của chúng ta, để minh họa cho các chế độ chọn và sự kết hợp giữa chúng, chúng ta sẽ sửa đoạn mã trong lớp SimpleTableTest ở file SimpleTableTest.java như sau:

```
public class SimpleTableTest extends JFrame {  
    protected JTable table;  
  
    public SimpleTableTest() {  
        [...]  
        pane.add(jsp, BorderLayout.CENTER);  
  
        JPanel outerPanel = new JPanel();  
        outerPanel.setLayout(new GridLayout(1, 2, 0, 0));  
        JPanel innerPanel = new JPanel();  
        innerPanel.setLayout(new FlowLayout());  
        JCheckBox modeBox = new JCheckBox("Row", true);  
        modeBox.addItemListener(new ItemListener() {
```

```

        public void itemStateChanged(ItemEvent event) {
            JCheckBox box = (JCheckBox)(event.getSource());
            table.setRowSelectionAllowed(box.isSelected());
        }
    });
    innerPanel.add(modeBox);
    modeBox = new JCheckBox("Column");
    modeBox.addItemListener(new ItemListener() {

        public void itemStateChanged(ItemEvent event) {
            JCheckBox box = (JCheckBox)(event.getSource());
            table.setColumnSelectionAllowed(box.isSelected());
        }
    });
    innerPanel.add(modeBox);
    modeBox = new JCheckBox("Cell");
    modeBox.addItemListener(new ItemListener() {

        public void itemStateChanged(ItemEvent event) {
            JCheckBox box = (JCheckBox)(event.getSource());
            table.setCellSelectionEnabled(box.isSelected());
        }
    });
    innerPanel.add(modeBox);

    BevelBorder bb = new BevelBorder(BevelBorder.RAISED);
    TitledBorder tb = new TitledBorder(bb, "Table Selection Types");
    innerPanel.setBorder(tb);
    outerPanel.add(innerPanel);

    innerPanel = new JPanel();
    innerPanel.setLayout(new FlowLayout());

```

```

JComboBox listModes = new JComboBox();
listModes.addItem("Single Selection");
listModes.addItem("Single Interval Selection");
listModes.addItem("Multiple Interval Selections");
listModes.setSelectedIndex(2);
listModes.addItemListener(new ItemListener() {

    public void itemStateChanged(ItemEvent event) {
        JComboBox box = (JComboBox)(event.getSource());
        int index = box.getSelectedIndex();
        switch(index){
            case 0:
                table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
                break;
            case 1:

table.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
                break;
            case 2:

table.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
                break;
        }
    }
});
innerPanel.add(listModes);
bb = new BevelBorder(BevelBorder.RAISED);
tb = new TitledBorder(bb, "List Selection Modes");
innerPanel.setBorder(tb);
outerPanel.add(innerPanel);

pane.add(outerPanel, BorderLayout.SOUTH);
}

```

```

public static void main(String [] args){
    [...]
}
}

```

Bây giờ chạy lại chương trình và chúng ta có thể từ từ kiểm tra các chế độ chọn và tổ hợp của chúng có ảnh hưởng như thế nào đến việc chọn dữ liệu trong bảng.



Thiết lập việc chọn bằng dòng lệnh

Ngoài các sự kiện do người sử dụng có thể sinh ra để thay đổi việc chọn các ô trong một bảng, JTable cũng cung cấp cho chúng ta một số các phương thức để chúng ta có thể lấy thông tin và sinh ra các sự kiện bằng dòng lệnh. Các phương thức sẽ được liệt kê trong bảng dưới đây:

Phương thức	Thực hiện
<code>getSelectedRowCount()</code>	Trả về số hàng đang được chọn trong bảng
<code>getSelectedRows()</code>	Trả về một mảng các số nguyên, mỗi một số nguyên chính là chỉ số của một hàng đang được chọn trong bảng.
<code>getSelectedRow()</code>	Trả về một số nguyên, số nguyên này là chỉ số nhỏ nhất của một hàng trong tập hợp các hàng đang được chọn của bảng.
<code>setRowSelectionInterval(int index0, int index1)</code>	Chọn một tập các hàng có chỉ số từ index0 cho đến index1. Các hàng khác có chỉ số

	không thuộc vào khoảng này sẽ không được chọn kể cả khi chúng đã được chọn trước đó.
<code>addRowSelectionInterval(int index0,int index1)</code>	Chọn một tập các hàng có chỉ số từ index0 cho đến index1.
<code>getSelectedColumnCount()</code>	Trả về số cột đang được chọn trong một bảng
<code>getSelectedColumns()</code>	Trả về một mảng các số nguyên, mỗi số nguyên là một chỉ số của một cột đang được chọn trong bảng
<code>getSelectedColumn()</code>	Trả về một số nguyên là chỉ số nhỏ nhất của một cột trong số các cột đang được chọn của bảng.
<code>setColumnSelectionInterval(int index0, int index1)</code>	Chọn các cột có chỉ số nằm trong khoảng từ index0 đến index1. Các cột khác có chỉ số nằm ngoài khoảng này sẽ không được chọn bất kể trước đó chúng đã được chọn hay chưa.
<code>addColumnSelectionInterval(int index0, int index1)</code>	Chọn các cột có chỉ số nằm trong khoảng từ index0 đến index1

Tất cả các phương thức này được định nghĩa trong `JTable`. Khi một phương thức được gọi nó sẽ chuyển yêu cầu đến một `ListSelectionModel` để xử lý và lấy thông tin ở đó. Một cách cụ thể, các phương thức có liên quan đến việc chọn theo hàng sẽ chuyển yêu cầu về một model được quản lý bởi `JTable`, trong khi đó các phương thức liên quan đến việc chọn theo cột sẽ chuyển yêu cầu về một model được quản lý bởi một cài đặt của `TableColumnModel` trong bảng.

Như vậy kết thúc phần này, chúng ta đã nắm được các chế độ chọn khi thao tác với hàng và cột. Trong phần 6 chúng ta sẽ tìm hiểu về các thao tác với tiêu đề của bảng.

PHẦN 6

Như đã giới thiệu ở **PHẦN 5**, trong phần này, chúng ta sẽ học cách sử dụng và thao tác với các tiêu đề trong một bảng. Mỗi cột trong một bảng đều cần phải có một tiêu đề để giúp người sử dụng hình dung ra các giá trị trong cột đó là để biểu hiện cái gì. Trong **PHẦN 2** của loạt bài viết về JTable này, chúng ta đã được biết làm sao để tạo tiêu đề cho các cột trong bảng và phần này, sẽ bổ sung thêm những điều thú vị khi thao tác với các tiêu đề trong một bảng.

Việc tạo ra tiêu đề cho các cột trong một bảng không còn xa lạ với chúng ta. Tuy nhiên việc tạo ra các tiêu đề như thế chỉ giúp về mặt hiển thị và không hề có một chức năng gì khác nữa. Chẳng hạn như chúng chẳng có phản ứng gì khi chúng ta nhấp chuột vào chúng, hoặc đoạn chữ cho tiêu đề chỉ có thể nằm trên một dòng duy nhất.

Khi thiết kế giao diện, sẽ có lúc chúng ta cần đòi hỏi các chức năng khác nữa từ các tiêu đề này. Cho ví dụ, chúng ta có thể làm một tooltip để miêu tả rõ hơn về mỗi tiêu đề, hoặc cho phép một cột có thể được chọn và tự động sắp xếp theo thứ tự giảm dần hoặc tăng dần của các giá trị trong một cột bằng việc nhấp chuột vào tiêu đề đó, hoặc cho phép đoạn chữ trong tiêu đề cho thể được ngắt ra làm nhiều dòng. Các chức năng này hoàn toàn có thể làm được nếu chúng ta đã nắm vững về cách làm việc của các tiêu đề trong JTable.

Vẽ tiêu đề

Ô tiêu đề cho một cột trong bảng cũng giống như các ô dữ liệu khác, được vẽ nên bởi các renderer. Renderer mặc định cho một ô trong bảng sẽ kế thừa từ một JLabel và ô tiêu đề cũng không ngoại lệ. Một renderer cho một ô tiêu đề của một cột có thể được lấy ra từ phương thức **getHeaderRenderer()** của đối tượng TableColumn tương ứng với cột đó. Cho ví dụ, đoạn mã sau sẽ lấy ra một tham chiếu đến renderer của tiêu đề cho cột thứ hai trong một bảng:

```
JTable table;  
[...]  
TableColumnModel tcm = table.getColumnModel();  
TableColumn tc = tcm.getColumn(1);  
TableCellRenderer tcr = tc.getHeaderRenderer();
```

Ngoài khả năng có thể lấy về một renderer cho tiêu đề của một cột nhất định, chúng ta còn có thể chỉ định một renderer cụ thể cho một tiêu đề của cột đó. Trong ví dụ của chúng ta, nếu chúng ta muốn tạo một renderer để giúp đoạn chữ trong tiêu đề của một cột có thể hiển thị trên nhiều dòng khác nhau, thì khi đó renderer phải kế thừa một JPanel chứa nhiều JLabel thành các dòng thay vì chỉ một JLabel như mặc định. Bởi vì JLabel không hỗ trợ viết một đoạn chữ trên nhiều dòng giống như JPanel. Như vậy, đầu tiên ta phải tạo một renderer có tên là **MultiLineHeaderRenderer** nằm trong file **MultiLineHeaderRenderer.java** như sau:

```
public class MultiLineHeaderRenderer extends JPanel implements TableCellRenderer {

    public Component getTableCellRendererComponent(JTable table, Object value,
                                                    boolean isSelected, boolean hasFocus, int row, int column) {
        JLabel label;
        removeAll();

        StringTokenizer strtok = new StringTokenizer((String)value, "\r\n");
        setLayout(new GridLayout(strtok.countTokens(), 1));
        while(strtok.hasMoreElements()){
            label = new JLabel((String)strtok.nextElement(), JLabel.CENTER);
            LookAndFeel.installColorsAndFont(label, "TableHeader.background",
                                             "TableHeader.foreground", "TableHeader.font");
            add(label);
        }
        LookAndFeel.installBorder(this, "TableHeader.cellBorder");

        return this;
    }
}
```


Renderer trên yêu cầu tên tiêu đề của cột phải chứa các kí tự như ‘\r’ (carriage return) hoặc ‘\n’ (linefeed) ở những chỗ muốn xuống dòng. Cho ví dụ, ta thay đổi tiêu đề các cột trong lớp TableValues như sau:

```
public class TableValues extends AbstractTableModel{

    [...]
    public final static boolean GENDER_FEMALE = false;
    public final static String[] columnNames = {
        "First Name", "Last Name", "Date of Birth", "Account\nBalance", "Gender"
    };

    public Object[][] values = {
        [...]
    };

    [...]
}
```

Bây giờ ta đặt cái MultiLineHeaderRenderer vừa tạo ở trên làm renderer cho tiêu đề của các cột trong bảng. Việc này thực hiện trong lớp SimpleTableTest như sau:

```
public class SimpleTableTest extends JFrame{
    protected JTable table;

    public SimpleTableTest(){
        [...]
        TableColumnModel tcm = table.getColumnModel();
        TableColumn tc = tcm.getColumn(TableValues.GENDER);
        tc.setCellRenderer(new GenderRenderer());
        tc.setCellEditor(new GenderEditor());
        MultiLineHeaderRenderer mlhr = new MultiLineHeaderRenderer();
        tc = tcm.getColumn(TableValues.ACCOUNT_BALANCE);
        tc.setHeaderRenderer(mlhr);
        table.setDefaultRenderer(Float.class, new CurrencyRenderer());
    }
}
```

```

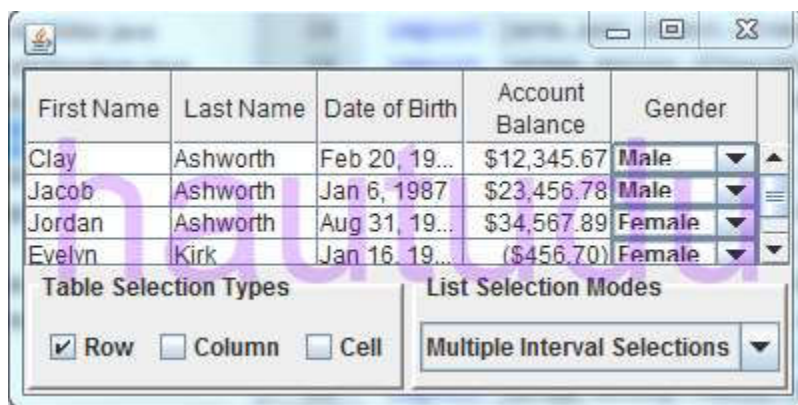
JScrollPane jsp = new JScrollPane(table);
pane.add(jsp, BorderLayout.CENTER);

[...]
}

public static void main(String [] args){
    [...]
}
}

```

Giờ chạy chương trình chúng ta có thể thấy tiêu đề của cột Account Balance sẽ được ngắt thành hai dòng như hình dưới đây:



Đặt Tooltip cho tiêu đề

Trong phần này chúng ta sẽ học cách thêm tooltip vào trong tiêu đề của các cột trong bảng. Bởi vì chúng ta thường cho renderer kế thừa từ các thành phần hiển thị, các thành phần hiển thị thì được kế thừa từ lớp **JComponent**, vì thế chúng ta có thể hoàn toàn gọi phương thức **setToolTipText()** từ renderer của một tiêu đề để đặt tooltip cho tiêu đề đó. Trở lại ví dụ, giờ chúng ta muốn đặt tooltip cho tiêu đề của cột Account Balance, ta chỉ cần sửa lớp SimpleTableTest như sau:

```

public class SimpleTableTest extends JFrame{
    protected JTable table;

    public SimpleTableTest(){

```

```

[...]  

MultiLineHeaderRenderer mlhr = new MultiLineHeaderRenderer();  

mlhr.setToolTipText("This is the person's current account balance");  

tc = tcm.getColumn(TableValues.ACCOUNT_BALANCE);  

[...]  

}  
  

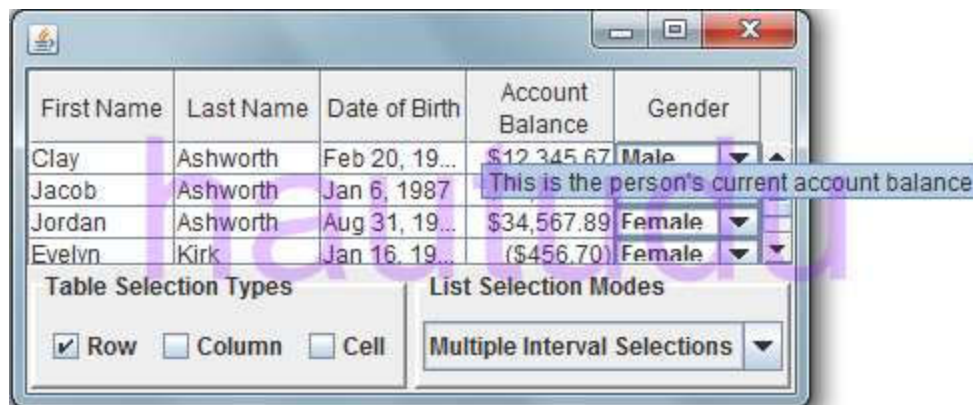
public static void main(String [] args){  

    [...]  

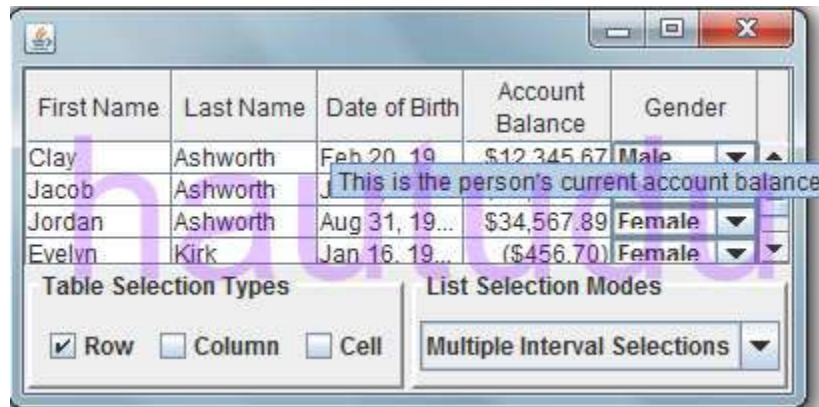
}

```

Giờ chúng ta chạy chương trình, khi di chuyển chuột qua tiêu đề của cột Account Balance, tooltip cho cột này sẽ hiện lên như hình dưới đây:



Ở đây, chúng ta sẽ nhận ra một vấn đề có liên quan đến sự chỉ định tooltip cho một cột. Ở ví dụ của chúng ta, chúng ta chỉ gán một renderer cho một cột duy nhất. Tuy nhiên trong thực tế thì một renderer có thể được gán cho nhiều cột trong một bảng. Hay nói cách khác, một renderer có thể được sử dụng để vẽ cho rất nhiều ô trong bảng và điều này cũng đúng cho các renderer của tiêu đề các cột. Trong ví dụ, một đối tượng của lớp `MultiLineHeaderRenderer` có thể được sử dụng để vẽ cho các tiêu đề của cả cột Account Balance lẫn cột Date of Birth. Tuy nhiên, việc đặt tooltip thì chỉ có thể là duy nhất cho một cột mà thôi. Nếu như ta cũng sử dụng lại renderer của cột Account Balance cho cột Date of Birth thì tooltip của cột Account Balance cũng trở thành tooltip của cột Date of Birth và như vậy thì hoàn toàn không thích hợp như hình dưới đây:



Có rất nhiều cách để khắc phục vấn đề này. Một trong những cách dễ nhất là gán cho các cột các đối tượng renderer khác nhau. Chẳng hạn, trong ví dụ chúng ta sẽ gán renderer là hai đối tượng khác nhau của lớp `MultiLineHeaderRenderer` cho hai cột `Account Balance` và `Date of Birth`. Ngoài ra, chúng ta còn có cách khác sẽ được miêu tả sau.

JTableHeader

Trở lại với phần 1 của loạt bài viết này, khi đó ta chạy chương trình ví dụ, thì bảng hiện lên không có tiêu đề cho các cột. Vấn đề này được khắc phục trong phần 2 khi mà `JTable` của chúng ta được thêm vào một `JScrollPane` và phương thức `getColumnName()` trong model được cài đặt để trả về tiêu đề cho các cột trong bảng. Tiêu đề của các cột sẽ tự động hiển thị khi mà bảng của chúng ta được hiển thị trong một `JScrollPane` và cụ thể hơn, chúng được hiển thị trong vùng gọi là **column header viewport** của cửa sổ cuộn. Column header viewport là phần không gian ở bên trên phần không gian chính của một `JScrollPane`, và tất nhiên, chúng ta hoàn toàn có thể truy cập, chỉnh sửa các thành phần được hiển thị trong vùng này sử dụng hai phương thức là `getColumnHeader()` và `setColumnHeader()` của `JScrollPane`. Cho ví dụ, giả sử chúng ta muốn hiển thị một nút bấm `JButton` trong phần column header viewport của cửa sổ cuộn trong ví dụ, chúng ta có thể tạm thời sửa lớp `SimpleTableTest` như sau:

```
public class SimpleTableTest extends JFrame {
    protected JTable table;

    public SimpleTableTest() {
        [...]
        table.setDefaultRenderer(Float.class, new CurrencyRenderer());
    }
}
```

```

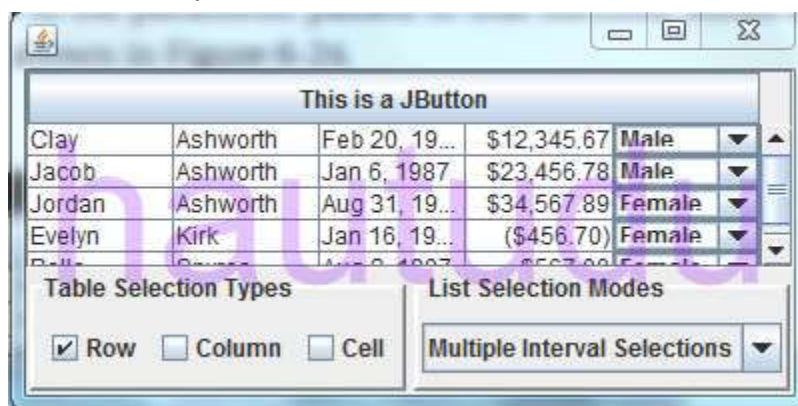
JScrollPane jsp = new JScrollPane(table){
    public void setColumnHeaderView(Component comp){
        super.setColumnHeaderView(new JButton("This is a JButton"));
    }
};
pane.add(jsp, BorderLayout.CENTER);

[...]
}

public static void main(String [] args){
    [...]
}
}

```

Đoạn mã trên đã cài đặt lại phương thức **setColumnHeaderView()** JScrollPane và điều này dẫn tới việc một JButton được hiển thị trong phần column header viewport của JScrollPane như hình dưới đây:



Mặc dù chúng ta thường không có nhu cầu để sử dụng phương thức này, nhưng nó chỉ ra rằng, chúng ta có thể sử dụng bất cứ thành phần nào để làm thành một tiêu đề. Một câu hỏi đặt ra đó là, nếu chúng ta không chỉ ra rõ ràng như ở ví dụ trên thì mặc định thành phần nào sẽ được sử dụng? Câu trả lời đó là một đối tượng của lớp **JTableHeader**.

JTableHeader là một thành phần hiển thị để cung cấp cho chúng ta các hành vi tương tác liên quan đến việc di chuyển và thay đổi kích thước các cột của một JTable. Cho ví dụ,

trong phần 2 của loạt bài viết, chúng ta có đề cập về việc thay đổi kích thước của một cột trong bảng, và đây chính là chúng ta đang tương tác với một đối tượng của `JTableHeader`. Ngoài ra, chúng ta còn có thể sắp xếp lại các cột vào các vị trí mà chúng ta muốn bằng các nhấp chuột lên từng cột đó, giữ chuột trái rồi kéo thả đến vị trí chúng ta muốn. Và đó cũng là một trong những chức năng mà `JTableHeader` cung cấp cho chúng ta.

Một trong những chức năng khác của `JTableHeader` là trả về đoạn chữ tooltip khi mà chúng ta di chuột lên trên tiêu đề của một cột nào đó trong bảng. Bên trên chúng ta đã nói về việc sinh ra tooltip cho một cột là nhờ các renderer, và chúng ta cũng có thể ngạc nhiên là tại sao `JTableHeader` ở đây lại sinh ra tooltip nữa? Nhưng thực vậy, chính là `JTableHeader` sinh ra tooltip cho một cột trong bảng nhưng để lấy tooltip đó, nó phải chuyển yêu cầu đến renderer của cột tương ứng.

Khi phương thức `getToolTipText()` của `JTableHeader` được gọi, nó được truyền vào một tham số đầu vào là một `MouseEvent` để cho phép `JTableHeader` xác định được tiêu đề của cột nào đang được con trỏ chuột di chuyển qua. Sau đó, `JTableHeader` chọn ra renderer tương ứng với cột đó để lấy đoạn chữ tooltip trả về từ renderer đó. Cách này có vẻ tốt khi mà mỗi tiêu đề cho mỗi cột trong bảng có một renderer của riêng nó, nhưng sẽ không thích hợp khi mà một renderer được gán cho nhiều cột. Để khắc phục vấn đề này, ta tạo một lớp kế thừa từ `JTableHeader` và trong nó duy trì một mảng gồm các tooltip, sau đó trả về tooltip lấy ra từ mảng đó thay vì phải gọi đến các renderer. Chúng ta tạo lớp `JTableHeaderToolTips` trong file `JTableHeaderToolTip.java` như sau:

```
public class JTableHeaderToolTips extends JTableHeader{
    protected String[] toolTips;
    public JTableHeaderToolTips(TableColumnModel tcm){
        super(tcm);
    }

    public void setToolTips(String [] tips){
        toolTips = tips;
    }

    @Override
    public String getToolTipText(MouseEvent event){
        String tip = super.getToolTipText(event);
```

```

        int column = columnAtPoint(event.getPoint());
        if((toolTips != null) && (column < toolTips.length) && (toolTips[column] != null)){
            tip = toolTips[column];
        }
        return tip;
    }
}

```

Bây giờ chúng ta sử dụng lớp `JTableHeaderToolTips` trong lớp `SimpleTableTest` như sau:

```

public class SimpleTableTest extends JFrame{
    protected JTable table;
    public SimpleTableTest(){
        [...]
        MultiLineHeaderRenderer mlhr = new MultiLineHeaderRenderer();
        tc = tcm.getColumn(TableValues.ACCOUNT_BALANCE);
        tc.setHeaderRenderer(mlhr);
        JTableHeaderToolTips jthtt = new
JTableHeaderToolTips(table.getColumnModel());
        jthtt.setToolTips(new String[]{"Customer's First Name", "Customer's Last
Name",
        "Customer's Date of Birth", "Customer's Account Balance", "Customer's
Gender"});
        table.setTableHeader(jthtt);
        table.setDefaultRenderer(Float.class, new CurrencyRenderer());
        JScrollPane jsp = new JScrollPane(table);
        pane.add(jsp, BorderLayout.CENTER);

        [...]
    }
    public static void main(String [] args){
        [...]
    }
}

```


Giờ chạy chương trình chúng ta sẽ có được các tooltip trên các tiêu đề của các cột:



Chúng ta có thể làm như trên đó là nhờ khả năng xác định được cột nào đang được con trỏ chuột di chuyển qua. Để làm như vậy chúng ta phải sử dụng phương thức **columnAtPoint()** được định nghĩa trong **JTableHeader**. Chúng ta cũng có thể sử dụng kỹ thuật trên để hiển thị tooltip cho các ô khác trong bảng. Các bước cũng tương tự, để làm tooltip cho các ô khác thì chúng ta phải cài đặt lại phương thức **getToolTipText()** của **JTableHeader**, sử dụng hai phương thức là **rowAtPoint()** và **columnAtPoint()** để nhận diện ra vị trí của ô đang được con trỏ chuột di chuyển qua.

Một trường hợp khác mà chúng ta vẫn phải sử dụng đến **JTableHeader** đó là để phát hiện và xử lý các sự kiện chuột xảy ra trên các tiêu đề. Cho ví dụ, giả sử ứng dụng của chúng ta cho phép người sử dụng chọn một cột của bảng bằng việc kích vào tiêu đề của cột đó. Mặc định thì **JTable** không cho phép chúng ta làm điều này, vì vậy, chúng ta sẽ phải tự tay bắt và xử lý sự kiện nhấp chuột. Trong trường hợp yêu cầu cho tooltip, yêu cầu này sẽ được chuyển đến renderer của cột tương ứng. Tuy nhiên, trong trường hợp này của chúng ta thì không giống như vậy, để có thể bắt được sự kiện chuột, chúng ta phải đăng ký một listener cho **JTableHeader**. Chúng ta chỉnh sửa lớp **SimpleTableTest** trong file **SimpleTableTest.java** như sau:

```
public class SimpleTableTest extends JFrame{
    protected JTable table;

    public SimpleTableTest(){
        [...]
        table = new JTable(tv);
        table.setRowSelectionAllowed(false);
    }
}
```



```

table.setColumnSelectionAllowed(true);
TableModel tcm = table.getColumnModel();
[...]
JScrollPane jsp = new JScrollPane(table);
pane.add(jsp, BorderLayout.CENTER);
addHeaderListener();

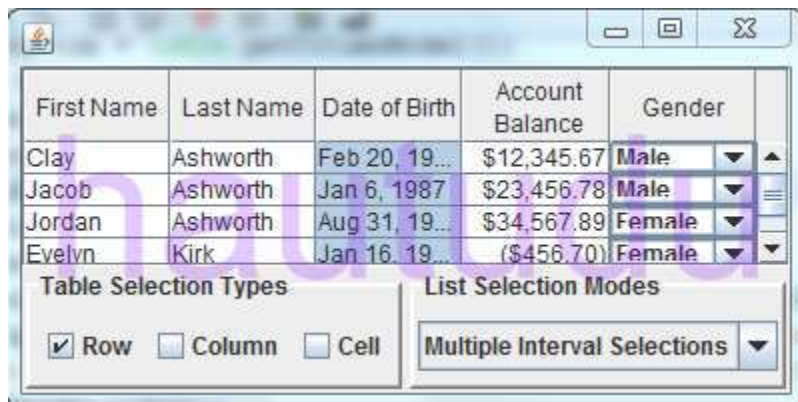
JPanel outerPanel = new JPanel();
[...]
}

public void addHeaderListener(){
    table.getTableHeader().addMouseListener(new MouseAdapter(){
        public void mousePressed(MouseEvent event){
            JTableHeader header = (JTableHeader)(event.getSource());
            int index = header.columnAtPoint(event.getPoint());
            table.setColumnSelectionInterval(index, index);
        }
    });
}

public static void main(String [] args){
    [...]
}
}

```

Bây giờ chạy chương trình, khi chúng ta nhấp chuột vào tiêu đề của cột Date of Birth, cả cột đó sẽ được chọn như hình dưới đây:



First Name	Last Name	Date of Birth	Account Balance	Gender
Clay	Ashworth	Feb 20, 19...	\$12,345.67	Male
Jacob	Ashworth	Jan 6, 1987	\$23,456.78	Male
Jordan	Ashworth	Aug 31, 19...	\$34,567.89	Female
Evelyn	Kirk	Jan 16, 19...	(\$456.70)	Female

Table Selection Types
☒ Row
 ☐ Column
 ☐ Cell

List Selection Modes

Multiple Interval Selections

Như vậy, trong phần này chúng ta đã tìm hiểu về các thao tác khi làm việc với tiêu đề của các cột trong bảng. Chúng ta có thể tùy biến thiết kế tiêu đề và làm cho nó thêm phong phú với các tính năng tương tác với người dùng qua các sự kiện chuột. Trong phần 7, chúng ta sẽ tiếp tục học cách để tạo một tiêu đề cho các hàng trong bảng và tạo một cột đóng băng.

PHẦN 7

Trong phần này chúng ta sẽ học cách để tạo tiêu đề cho các hàng trong một bảng và tạo cột đóng băng.

Tạo tiêu đề cho hàng

Trong rất nhiều trường hợp, một bảng chỉ chứa các tiêu đề cho các cột là chưa đủ bởi vì sẽ có lúc chúng ta còn muốn tạo tiêu đề cho các hàng trong bảng nữa. Để làm điều này thì cũng khá là dễ dàng bởi vì JScrollPane không chỉ có viewport cho tiêu đề của cột mà còn cả cho tiêu đề của hàng nữa. Không giống như viewport cho tiêu đề của cột, theo mặc định, viewport cho tiêu đề của hàng là trống. Tuy nhiên, như đã nói ở trên, việc tạo ra một tiêu đề cho hàng và hiển thị nó là khá dễ dàng.

Dưới đây chúng ta tạo một lớp **RowNumberHeader** trong file **RowNumberHeader.java** để làm một tiêu đề cho hàng trong bảng ở ví dụ của chúng ta. Bản chất của tiêu đề dưới đây chính là một JTable, và model cho JTable này chính là lớp **RowNumberHeaderModel**. Số cột của bảng này là 1, số hàng bằng với số hàng của bảng mà ta đang muốn thêm tiêu đề cho hàng. Giá trị trong từng ô của bảng bằng với giá trị của chỉ số hàng tương ứng cộng thêm một đơn vị.

```
public class RowNumberHeader extends JTable{
    protected JTable mainTable;

    public RowNumberHeader (JTable table){
        super();
        mainTable = table;
        setModel(new RowNumberTableModel());
        setPreferredScrollableViewportSize(getMinimumSize());
        setRowSelectionAllowed(false);
        JComponent renderer = (JComponent)getDefaultRenderer(Object.class);
        LookAndFeel.installColorsAndFont(renderer, "TableHeader.background",
                                          "TableHeader.foreground", "TableHeader.font");
        LookAndFeel.installBorder(this, "TableHeader.cellBorder");
    }

    @Override
```

```

public int getRowHeight(int row){
    return mainTable.getRowHeight();
}

class RowNumberTableModel extends AbstractTableModel{

    public int getRowCount() {
        return mainTable.getModel().getRowCount();
    }

    public int getColumnCount() {
        return 1;
    }

    public Object getValueAt(int rowIndex, int columnIndex) {
        return new Integer(rowIndex+1);
    }
}
}

```

Giờ chúng ta thay đổi lớp SimpleTableTest như sau:

```

public class SimpleTableTest extends JFrame{
    protected JTable table;

    public SimpleTableTest(){
        [...]
        JScrollPane jsp = new JScrollPane(table);
        JViewport jvp = new JViewport();
        jvp.setView(new RowNumberHeader(table));
        jvp.setRowHeader(jvp);
        pane.add(jsp, BorderLayout.CENTER);
        addHeaderListener();
    }
}

```

```


    [...]
}

public void addHeaderListener(){
    [...]
}

public static void main(String [] args){
    [...]
}
}

```

Bây giờ chạy chương trình, ta sẽ tiêu đề của hàng hiện lên là số thứ tự của các hàng như hình dưới đây:



	First Name	Last Name	Date of Birth	Account Balance	Gender
1	Clay	Ashworth	Feb 20, 19...	\$12,345.67	Male
2	Jacob	Ashworth	Jan 6, 1987	\$23,456.78	Male
3	Jordan	Ashworth	Aug 31, 19...	\$34,567.89	Female
4	Evelyn	Kirk	Jan 16, 19...	(\$456.70)	Female

Table Selection Types

☒ Row ☐ Column ☐ Cell

List Selection Modes

Multiple Interval Selections

Tạo cột đóng băng

Ngoài việc hiển thị tiêu đề cho hàng, sẽ có lúc chúng ta còn muốn “đóng băng” một hay nhiều cột trong một bảng lại sao cho chúng vẫn ở nguyên vị trí kể cả khi chúng ta cuộn thanh cuộn ngang trong bảng. Trong ví dụ của chúng ta, chúng ta có thể đóng băng cột đầu tiên (cột First Name) sao cho nó luôn luôn có thể nhìn thấy khi mà chúng ta cuộn thanh trượt ngang của bảng. Các bước để làm cột đóng băng sẽ như sau:

1. Tạo một JTable và chúng ta gọi nó là **bảng chính**. Dùng một JScrollPane để bọc JTable này lại. Bảng chính này có nhiệm vụ hiển thị những dữ liệu không bị đóng băng.

2. Tạo một JTable thứ hai, chúng ta gọi nó là **bảng tiêu đề** và cũng thêm nó vào JScrollPane. Bảng này sẽ sử dụng TableModel giống của bảng chính nhưng sẽ để hiển thị các cột bị đóng băng
3. Tạo một TableColumnModel để sau này sẽ gán nó vào bảng tiêu đề
4. Xóa các đối tượng của TableColumn tương ứng với các cột bị đóng băng trong TableColumnModel của bảng chính, và sau đó thêm chúng vào TableColumnModel của bảng tiêu đề mà chúng ta đã tạo ở bước 3
5. Gán TableColumnModel chúng ta tạo ở bước 3 vào bảng tiêu đề sử dụng phương thức setColumnModel()
6. JScrollPane chứa bảng tiêu đề bây giờ cũng chứa một JTableHeader của bảng tiêu đề đó ở trong column header viewport của nó. Lấy một tham chiếu đến JTableHeader này, và chuyển nó đến góc trên bên trái của JScrollPane cái có chứa bảng chính. Chúng ta có thể làm điều này sử dụng phương thức **setCorner()** của JScrollPane có chứa bảng chính.
7. Đặt độ rộng ưa thích cho viewport của bảng tiêu đề sao cho nó chỉ đủ rộng để hiển thị các cột bị đóng băng. Độ rộng mặc định của nó là 450, và độ rộng này luôn lớn hơn độ rộng cần thiết.

Tóm lại, để đóng băng các cột trong một bảng, chúng ta chia bảng ra thành hai bảng con. Một bảng chính để hiển thị các cột không đóng băng. Bảng tiêu đề còn lại để hiển thị các cột đóng băng và đóng vai trò như một tiêu đề cho hàng trong cửa sổ cuộn có chứa bảng chính. Tiêu đề cột của bảng tiêu đề này được chuyển đến góc trên bên trái của cửa sổ cuộn có chứa bảng chính. Chúng ta tạo một lớp là **FrozenColumnHeader** trong file **FrozenColumnHeader.java** như sau:

```
public class FrozenColumnHeader extends JScrollPane{
    protected JTable mainTable;
    protected JTable headerTable;
    protected int columnCount;

    public FrozenColumnHeader(JTable table, int columns){
        super();
        mainTable = table;
        headerTable = new JTable(mainTable.getModel());
        getViewPort().setView(headerTable);
    }
}
```

```

        columnCount = columns;
    }

    @Override
    public void addNotify(){
        TableColumn column;
        super.addNotify();
        TableColumnModel mainModel = mainTable.getColumnModel();
        TableColumnModel headerModel = new DefaultTableColumnModel();
        int frozenWidth = 0;
        for(int i = 0; i<columnCount; i++){
            column = mainModel.getColumn(0);
            mainModel.removeColumn(column);
            headerModel.addColumn(column);
            frozenWidth += column.getPreferredWidth() + headerModel.getColumnMargin();
        }

        headerTable.setColumnModel(headerModel);
        Component columnHeader = getColumnHeader().getView();
        getColumnHeader().setView(null);
        JScrollPane mainScrollPane =
            (JScrollPane)SwingUtilities.getAncestorOfClass(JScrollPane.class, mainTable);
        mainScrollPane.setCorner(JScrollPane.UPPER_LEFT_CORNER, columnHeader);
        headerTable.setPreferredScrollableViewportSize(new Dimension(frozenWidth, 0));
    }
}

```

Chúng ta có thể sử dụng lớp vừa tạo ở trên bằng cách tạo một đối tượng của nó và truyền vào constructor của nó một tham chiếu đến cái JTable là bảng chính của chúng ta, đồng thời một tham số là số lượng cột sẽ bị đóng băng (tính từ cột đầu tiên trong bảng chính). Trong ví dụ, ta có thể sửa lớp SimpleTableTest thành như sau:

```

public class SimpleTableTest extends JFrame{
    protected JTable table;

```

```

public SimpleTableTest(){
    [...]
    JScrollPane jsp = new JScrollPane(table);
    JViewport jvp = new JViewport();
    jvp.setView(new FrozenColumnHeader(table, 1));
    table.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
    jsp.setRowHeader(jvp);
    pane.add(jsp, BorderLayout.CENTER);
    addHeaderListener();

    [...]
}

public void addHeaderListener(){
    [...]
}

public static void main(String [] args){
    [...]
}
}

```

Bây giờ chạy chương trình chúng ta sẽ thấy rằng khi chúng ta cuộn thanh trượt ngang thì cột First Name bị đóng băng sẽ vẫn giữ nguyên vị trí của nó như hình dưới đây:



Mặc dù ví dụ trên chỉ minh họa làm thế nào để đóng băng một cột, nhưng chúng ta hoàn toàn có thể sử dụng kỹ thuật trên để đóng băng nhiều cột. Với các tiếp cận đó, chúng ta cũng có thể đóng băng các dòng dữ liệu trong một bảng bằng cách tạo ra một bảng có chứa các dòng cần đóng băng rồi thêm nó vào column header viewport của JScrollPane.

Trong phần 8, chúng ta sẽ tìm hiểu cách để sắp xếp các dòng trong một bảng theo chiều tăng dần hoặc giảm dần của các giá trị trong một cột nào đó của bảng.

PHẦN 8

Trong phần này, chúng ta sẽ tìm hiểu cách để sắp xếp các hàng trong một bảng theo chiều tăng dần hoặc giảm dần của các giá trị trong một cột nào đó của bảng.

Sắp xếp các hàng trong một bảng

Khi hiển thị thông tin trong một JTable, sẽ có lúc chúng ta muốn sắp xếp lại các hàng trong bảng đó theo chiều tăng dần hoặc giảm dần của các giá trị trong một hoặc nhiều cột của bảng. Bởi vì việc sắp xếp thường là chậm và sẽ trở thành phức tạp khi mà tập dữ liệu của chúng ta lớn, vì vậy chúng ta nên có các dữ liệu đã được sắp xếp sẵn từ một ứng dụng bên ngoài. Ví dụ như nếu chúng ta hiển thị dữ liệu lấy từ một cơ sở dữ liệu quan hệ, chúng ta có thể lấy được tập dữ liệu đã được sắp xếp sẵn từ câu truy vấn. Tuy nhiên, sẽ có đôi khi vì một lý do nào đó chúng ta cần phải sắp xếp dữ liệu trên chính cái bảng của chúng ta, và bởi vì JTable không trực tiếp hỗ trợ việc sắp xếp này, cho nên chúng ta phải tự tay viết những dòng lệnh để thực hiện việc này.

Để sắp xếp dữ liệu hiển thị trong một bảng, chúng ta có thể sử dụng một trong hai cách tiếp cận: Cách thứ nhất là sắp xếp dữ liệu “ngay tại chỗ”, cách thứ hai là việc sắp xếp được thực hiện thông qua một lớp nằm giữa JTable và TableModel. Sắp xếp dữ liệu ngay tại chỗ có nghĩa là chúng ta thay đổi vị trí của dữ liệu trong một mảng hoặc trong một đối tượng collection. Ví dụ như trong trường hợp của chúng ta, dữ liệu được định nghĩa trong lớp TableValues, chúng ta sẽ sắp xếp lại các giá trị này trong mảng sao cho chúng tăng dần hoặc giảm dần. Tức là trong cách tiếp cận này, ta sắp xếp lại dữ liệu trong cái model của bảng.

Một cách tiếp cận có vẻ linh động hơn đó là thêm một lớp làm việc sắp xếp giữa bảng và cái model của nó. Cụ thể hơn, nó liên quan đến việc tạo ra một TableModel thứ hai gọi là **model sắp xếp** (sort model). Model sắp xếp này sẽ có một tham chiếu đến cái **model gốc** (source model – trong ví dụ chính là TableValues) của chúng ta. Trong trường hợp này, dữ liệu trong model gốc không cần phải chuyển đổi hay thay đổi gì. Thay vào đó, model sắp xếp có thể tạo một danh sách các chỉ số. Các chỉ số này tham chiếu đến các dữ liệu trong model gốc và được sắp xếp theo thứ tự tăng dần hoặc giảm dần. Cho ví dụ, giả sử có ba giá trị kiểu String được lưu trong model gốc như sau:

- Kirk
- Ashworth
- Spyres

Model sắp xếp có thể sắp xếp các giá trị này và tạo một tập các chỉ số tham chiếu đến chúng. Trong trường hợp này, các chỉ số sẽ được đánh như sau:

- 1
- 0
- 2

Bằng việc sử dụng một danh sách các chỉ số để tham chiếu đến các hàng trong model gốc, dữ liệu có thể xuất hiện theo thứ tự đã sắp xếp, kể cả dù trong thực tế, dữ liệu vẫn được lưu theo thứ tự ban đầu của nó. Chúng ta tạo một lớp là **SortedTableModel** trong file **SortedTableModel.java** như sau:

```
public class SortedTableModel extends AbstractTableModel{

    protected TableModel sourceModel;
    protected int[] indexValues;

    public SortedTableModel(TableModel model){
        super();
        sourceModel = model;
    }

    public int getRowCount() {
        return sourceModel.getRowCount();
    }

    public int getColumnCount() {
        return sourceModel.getColumnCount();
    }

    public Object getValueAt(int rowIndex, int columnIndex) {
        if(indexValues != null){
            rowIndex = getSourceIndex(rowIndex);
        }
        return sourceModel.getValueAt(rowIndex, columnIndex);
    }
}
```

```
@Override
public void setValueAt(Object value, int row, int column){
    if(indexValues != null){
        row = getSourceIndex(row);
    }
}
```

```
@Override
public boolean isCellEditable(int row, int column){
    return sourceModel.isCellEditable(row, column);
}
```

```
@Override
public String getColumnName(int column){
    return sourceModel.getColumnName(column);
}
```

```
@Override
public Class getColumnClass(int column){
    return sourceModel.getColumnClass(column);
}
```

```
public int getSourceIndex(int index){
    if(indexValues != null){
        return indexValues[index];
    }
    return -1;
}
```

```
public void sortRows(int column, boolean ascending){
    SortedItemHolder holder;
    TreeSet sortedList = new TreeSet();
```

```
int count = getRowCount();
for(int i=0; i< count; i++){
    holder = new SortedItemHolder(sourceModel.getValueAt(i, column),i);
    sortedList.add(holder);
}
indexValues = new int[count];
Iterator iterator = sortedList.iterator();
int index = (ascending ? 0: count-1);
while(iterator.hasNext()){
    holder = (SortedItemHolder)(iterator.next());
    indexValues[index] = holder.position;
    index += (ascending ? 1: -1);
}
refreshViews();
}
```

```
public void clearSort(){
    indexValues = null;
    refreshViews();
}
```

```
public void refreshViews(){
    fireTableDataChanged();
}
```

```
class SortedItemHolder implements Comparable{

    public final Object value;
    public final int position;

    public SortedItemHolder(Object value, int position){
        this.value = value;
        this.position = position;
    }
}
```

```

    }

    public int compareTo(Object parm) {
        SortedItemHolder holder = (SortedItemHolder)parm;
        Comparable comp = (Comparable)value;
        int result = comp.compareTo(holder.value);
        if(result == 0){
            result = (position < holder.position) ? -1: 1;
        }
        return result;
    }

    @Override
    public int hashCode(){
        return position;
    }

    @Override
    public boolean equals(Object comp){
        if(comp instanceof SortedItemHolder){
            SortedItemHolder other = (SortedItemHolder)comp;
            if((position == other.position) && (value == other.value)){
                return true;
            }
        }
        return false;
    }
}

```

Phương thức `sortRows` được sử dụng để chỉ ra dữ liệu sẽ được sắp xếp theo cột nào và theo thứ tự là tăng dần hay giảm dần. Lớp **SortedItemHolder** nằm ở bên trong lớp `SortedTableModel` làm nhiệm vụ sắp xếp các giá trị và tất nhiên, bên trong lớp

SortedTableModel phải có một tham chiếu đến model gốc. Thêm vào đó, một phần của hai phương thức `getValueAt()` và `setValueAt()` là thực hiện việc chuyển đổi chỉ số hàng giữa hai model.

Bằng việc sử dụng lớp này, chúng ta có thể hiển thị dữ liệu trong bảng theo một thứ tự được sắp xếp. Chẳng hạn sửa đoạn mã trong lớp `SimpleTableTest` sau sẽ sắp xếp dữ liệu trong bảng theo giá trị tăng dần (từ trên xuống dưới) của cột Account Balance:

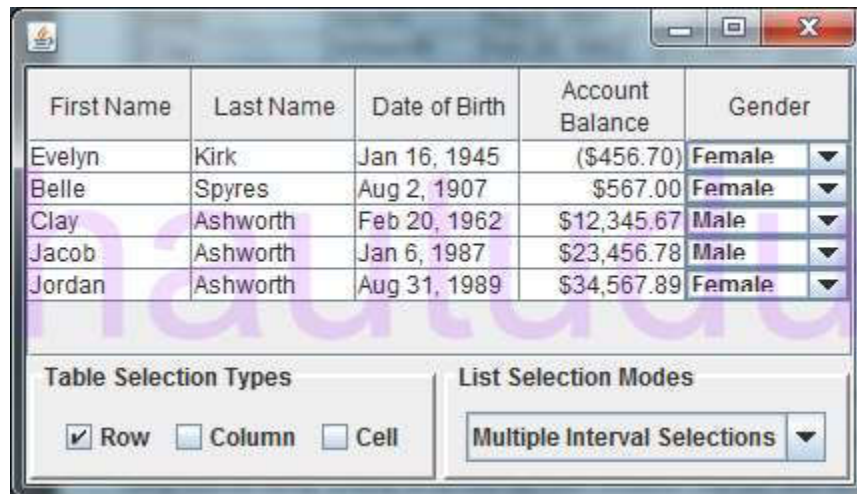
```
public class SimpleTableTest extends JFrame{
    protected JTable table;

    public SimpleTableTest(){
        Container pane = getContentPane();
        pane.setLayout(new BorderLayout());
        TableValues tv = new TableValues();
        SortedTableModel stm = new SortedTableModel(tv);
        stm.sortRows(TableValues.ACCOUNT_BALANCE, true);
        table = new JTable(stm);
        table.setRowSelectionAllowed(false);
        [...]
        JScrollPane jsp = new JScrollPane(table);
        pane.add(jsp, BorderLayout.CENTER);
        addHeaderListener();
        [...]
    }

    public void addHeaderListener(){
        [...]
    }

    public static void main(String [] args){
        [...]
    }
}
```

Kết quả khi chạy chương trình, các hàng trong bảng sẽ được sắp xếp lại sao cho giá trị trong cột Account Balance tăng dần như hình dưới đây:



First Name	Last Name	Date of Birth	Account Balance	Gender
Evelyn	Kirk	Jan 16, 1945	(\$456.70)	Female
Belle	Spyres	Aug 2, 1907	\$567.00	Female
Clay	Ashworth	Feb 20, 1962	\$12,345.67	Male
Jacob	Ashworth	Jan 6, 1987	\$23,456.78	Male
Jordan	Ashworth	Aug 31, 1989	\$34,567.89	Female

Table Selection Types: ☒ Row ☐ Column ☐ Cell

List Selection Modes: Multiple Interval Selections

Trong ví dụ trên, một cột được chọn để sắp xếp sẽ được cố định ở trong đoạn mã và chúng ta không thể thay đổi khi mà chương trình đã chạy. Tuy nhiên, việc tạo một giao diện để cho phép người sử dụng chọn một cột để sắp xếp theo thì cũng khá dễ dàng. Chúng ta có thể tạo một renderer cho các tiêu đề của các cột trong bảng. Renderer này sẽ phát hiện hiện tượng nhấp chuột, xác định cột nào đang được con trỏ chuột chỉ tới và sắp xếp các dữ liệu trong bảng dựa trên các giá trị của cột đó. Chúng ta tạo lớp **SortedColumnHeaderRenderer** trong file `SortedColumnHeaderRenderer` như sau:

```
public class SortedColumnHeaderRenderer implements TableCellRenderer{
    protected TableCellRenderer textRenderer;
    protected SortedTableModel sortedModel;
    protected int sortColumn = -1;
    protected boolean sortAscending = true;

    public SortedColumnHeaderRenderer(SortedTableModel model, TableCellRenderer
renderer){
        sortedModel = model;
        textRenderer = renderer;
    }

    public SortedColumnHeaderRenderer(SortedTableModel model){
```



```

        this(model, null);
    }

    public Component getTableCellRendererComponent(JTable table, Object value,
        boolean isSelected, boolean hasFocus, int row, int column) {
        Component text;
        JPanel panel = new JPanel();
        panel.setLayout(new BorderLayout());

        if(textRenderer != null){
            text = textRenderer.getTableCellRendererComponent(table, value, isSelected,
                hasFocus, row, column);
        }else{
            text = new JLabel((String)value, JLabel.CENTER);
            LookAndFeel.installColorsAndFont((JComponent)text,
                "TableHeader.background",
                "TableHeader.foreground", "TableHeader.font");
        }
        panel.add(text,BorderLayout.CENTER);

        if(column == sortColumn){
            BasicArrowButton bab = new
            BasicArrowButton((sortAscending?SwingConstants.NORTH:SwingConstants.SOUTH));
            panel.add(bab,BorderLayout.WEST);
        }
        LookAndFeel.installBorder(panel, "TableHeader.cellBorder");
        return panel;
    }

    public void columnSelected(int column){
        if(column!=sortColumn){
            sortColumn = column;
            sortAscending = true;

```

```

    }else{
        sortAscending = !sortAscending;
        if(sortAscending) sortColumn = -1;
    }
    if(sortColumn != -1){
        sortedModel.sortRows(sortColumn, sortAscending);
    }else{
        sortedModel.clearSort();
    }
}
}
}

```

Chúng ta cần chú ý đến hai điểm quan trọng trong renderer này. Thứ nhất, nó có thể được truyền vào một tham chiếu đến renderer khác. Renderer khác này có nhiệm vụ là vẽ đoạn chữ cho tiêu đề. Điều này cho phép chúng ta kết hợp được các chức năng của nhiều renderer lại với nhau. Nói cách khác, chúng ta vừa có thể tạo một bảng với các tiêu đề có thể hiển thị đoạn chữ trên nhiều dòng, vừa cho phép chúng ta chọn cột sắp xếp một cách động bằng cách nhấp chuột vào tiêu đề của cột đó.

Thứ hai, lớp này duy trì một biến để nhận diện ra cột nào sẽ được sắp xếp. Bởi vì thế, chúng ta chỉ có thể sử dụng một đối tượng duy nhất của lớp renderer này cho tất cả các cột mà chúng ta muốn chọn để sắp xếp trong bảng.

Khi chúng ta gán renderer này đến các ô tiêu đề, nó cho phép chúng ta có thể sắp xếp dữ liệu theo giá trị trong một cột bằng cách nhấp chuột vào tiêu đề của cột đó. Lần đầu tiên chúng ta kích vào tiêu đề của một cột, các hàng trong bảng sẽ được sắp xếp theo thứ tự tăng dần của giá trị trong cột đó. Nếu chúng ta kích chuột một lần nữa vào tiêu đề đó, các hàng sẽ được sắp xếp lại nhưng lần này là theo thứ tự giảm dần và lần kích chuột thứ ba sẽ là nguyên nhân để các hàng được trở về vị trí ban đầu khi chưa được sắp xếp. Khi mà giá trị trong bảng được sắp xếp, sẽ có một hiển thị xuất hiện để chỉ thị giá trị này được sắp xếp như thế nào. Đó là hình một mũi tên hướng lên trong trường hợp sắp xếp các giá trị theo thứ tự tăng dần và hình một mũi tên hướng xuống trong trường hợp sắp xếp các giá trị theo thứ tự giảm dần (từ trên xuống dưới). Chúng ta thay đổi lớp SimpleTableTest như sau:

```

public class SimpleTableTest extends JFrame{
    protected JTable table;

```

```
protected SortedColumnHeaderRenderer renderer;
```

```
public SimpleTableTest(){
    Container pane = getContentPane();
    pane.setLayout(new BorderLayout());
    TableValues tv = new TableValues();
    SortedTableModel stm = new SortedTableModel(tv);
    table = new JTable(stm);
    table.setRowSelectionAllowed(false);
    table.setColumnSelectionAllowed(true);
    TableColumnModel tcm = table.getColumnModel();
    TableColumn tc = tcm.getColumn(TableValues.GENDER);
    tc.setCellRenderer(new GenderRenderer());
    tc.setCellEditor(new GenderEditor());
    MultiLineHeaderRenderer mlhr = new MultiLineHeaderRenderer();
renderer = new SortedColumnHeaderRenderer(stm, mlhr);
int count = tcm.getColumnCount();
for(int i=0;i<count; i++){
    tc = tcm.getColumn(i);
    tc.setHeaderRenderer(renderer);
}
    JTableHeaderToolTips jthtt = new JTableHeaderToolTips(table.getColumnModel());
    [...]
    addHeaderListener();

    [...]
}
```

```
public void addHeaderListener(){
    table.getTableHeader().addMouseListener(new MouseAdapter(){
        public void mousePressed(MouseEvent event){
            JTableHeader header = (JTableHeader)(event.getSource());
            int index = header.columnAtPoint(event.getPoint());
```

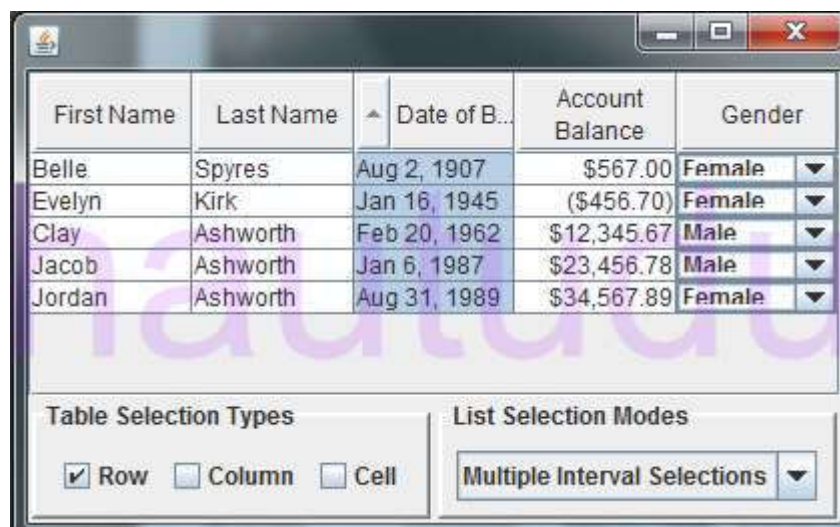
```

        renderer.columnSelected(index);
        table.setColumnSelectionInterval(index, index);
    }
});
}

public static void main(String [] args){
    [...]
}
}

```

Sau khi chạy chương trình và nhấp chuột lên cột Date of Birth, ta sẽ thấy các giá trị trong cột này được sắp xếp lại giống như hình dưới đây:



First Name	Last Name	Date of B...	Account Balance	Gender
Belle	Spyres	Aug 2, 1907	\$567.00	Female
Evelyn	Kirk	Jan 16, 1945	(\$456.70)	Female
Clay	Ashworth	Feb 20, 1962	\$12,345.67	Male
Jacob	Ashworth	Jan 6, 1987	\$23,456.78	Male
Jordan	Ashworth	Aug 31, 1989	\$34,567.89	Female

Table Selection Types: ☒ Row ☐ Column ☐ Cell

List Selection Modes: Multiple Interval Selections

Sử dụng interface Comparable

Một trong những giới hạn của cách sắp xếp mà chúng ta đã trình bày ở trên là việc sử dụng interface **Comparable** để xác định mối tương quan giữa giá trị của hai đối tượng (lớn hơn, nhỏ hơn, bằng nhau). Thông thường thì trong Java 2, hầu hết các lớp đều cài đặt interface Comparable và điều đó cho phép chúng ta có thể sắp xếp các đối tượng của chúng theo một cách thức nào đó. Cho ví dụ như các lớp Integer, Float, Long, String, Date... đều cài đặt interface Comparable. Tuy nhiên, lớp Boolean thì không cài đặt interface Comparable, bởi vì mặc dù tất nhiên là giá trị true không bằng với giá trị false, nhưng sẽ rất mập mờ trong việc quy định giữa true và false thì giá trị nào lớn hơn. Trong ví dụ của chúng ta, nếu chúng

ta nhấp chuột vào tiêu đề của cột Gender, chương trình sẽ sinh ra một **ClassCastException**. Đây là bởi vì chúng ta đang cố ép các giá trị thuộc kiểu Boolean về các giá trị kiểu Comparable.

Một trong những cách để giải quyết vấn đề trên đó là chúng ta sẽ kiểm tra kiểu dữ liệu trong cột được chọn trước khi các dữ liệu trong cột đó được sắp xếp. Việc lấy kiểu dữ liệu của một cột sẽ được thực hiện bởi phương thức **getColumnClass()** của TableModel và chúng ta có thể sử dụng đối tượng trả về thuộc kiểu Class của phương thức đó để xác định xem lớp của nó có cài đặt interface Comparable hay không. Trở lại ví dụ, chúng ta thực hiện việc này bằng việc thay đổi lớp SimpleTableTest như sau:

```
public class SimpleTableTest extends JFrame{
    protected JTable table;
    protected SortedColumnHeaderRenderer renderer;

    public SimpleTableTest(){
        [...]
    }

    public void addHeaderListener(){
        table.getTableHeader().addMouseListener(new MouseAdapter(){
            public void mousePressed(MouseEvent event){
                JTableHeader header = (JTableHeader)(event.getSource());
                int index = header.columnAtPoint(event.getPoint());
                Class dataType = table.getModel().getColumnClass(index);
                Class[] interfaces = dataType.getInterfaces();
                for(int i = 0; i<interfaces.length; i++){
                    if(interfaces[i].equals(java.lang.Comparable.class)){
                        renderer.columnSelected(index);
                        break;
                    }
                }
            }
        });
        table.setColumnSelectionInterval(index, index);
    }
}
```

```

    }

    public static void main(String [] args){
        [...]
    }
}

```

Khi chạy chương trình, với việc sửa đổi như trên giờ đây chúng ta không thể chọn sắp xếp theo giá trị trong cột Gender như hình dưới đây:



Tuy nhiên, vẫn còn một vấn đề nữa sinh ra từ việc làm này đó là khi ta chọn các cột như First Name hoặc Last Name, các giá trị cũng không được sắp xếp. Đây là bởi vì trong TableModel chúng ta chỉ chỉ định kiểu dữ liệu cho hai cột đó là Date of Birth và Account Balance. Các cột khác sẽ được ngầm hiểu rằng chúng có kiểu dữ liệu là Object. Hay nói cách khác, chương trình sẽ không thực hiện sắp xếp trên các cột First Name, Last Name, Gender bởi vì chương trình đang hiểu các cột này có kiểu dữ liệu là Object, mà kiểu Object thì lại không cài đặt interface Comparable. Tuy nhiên, vấn đề này có thể giải quyết một cách dễ dàng bằng cách chúng ta sửa lại phương thức getColumnClass() trong lớp TableValues như sau:

```

public class TableValues extends AbstractTableModel{

    public final static int FIRST_NAME = 0;

```

```
public final static int LAST_NAME = 1;
public final static int DATE_OF_BIRTH = 2;
public final static int ACCOUNT_BALANCE = 3;
public final static int GENDER = 4;

public final static boolean GENDER_MALE = true;
public final static boolean GENDER_FEMALE = false;
public final static String[] columnNames = {
    "First Name", "Last Name", "Date of Birth", "Account\nBalance", "Gender"
};

public Object[][] values = {
    [...]
};

public int getRowCount() {
    return values.length;
}

public int getColumnCount() {
    return values[0].length;
}

public Object getValueAt(int rowIndex, int columnIndex) {
    return values[rowIndex][columnIndex];
}

@Override
public String getColumnName(int column){
    return columnNames[column];
}

@Override
```

```
public Class getColumnClass(int column){
    Class dataType = super.getColumnClass(column);
    if(column == ACCOUNT_BALANCE){
        dataType = Float.class;
    }else if(column == DATE_OF_BIRTH){
        dataType = java.util.Date.class;
    }else if((column == FIRST_NAME) || (column == LAST_NAME)){
        dataType = String.class;
    }else if(column == GENDER){
        dataType = Boolean.class;
    }
    return dataType;
}
```

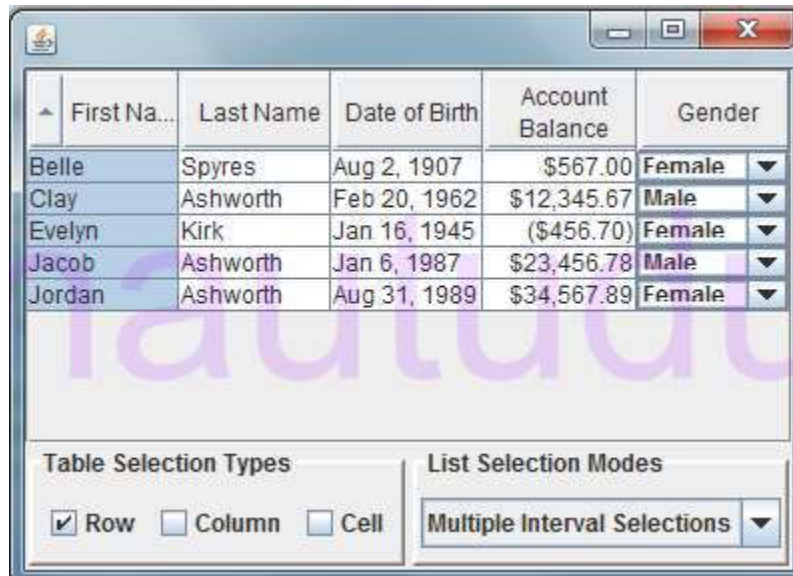
@Override

```
public boolean isCellEditable(int row, int column){
    if(column == GENDER){
        return true;
    }
    return false;
}
```

@Override

```
public void setValueAt(Object value, int row, int column){
    values[row][column] = value;
}
}
```


Bây giờ chạy chương trình, chúng ta có thể sắp xếp dữ liệu trong bảng dựa vào các giá trị của cột First Name như hình dưới đây:



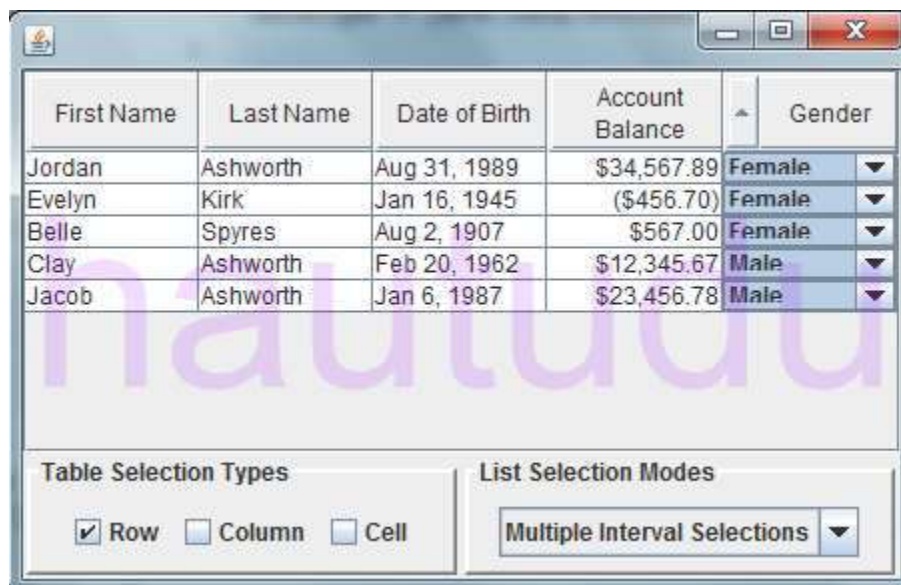
First Na...	Last Name	Date of Birth	Account Balance	Gender
Belle	Spyres	Aug 2, 1907	\$567.00	Female
Clay	Ashworth	Feb 20, 1962	\$12,345.67	Male
Evelyn	Kirk	Jan 16, 1945	(\$456.70)	Female
Jacob	Ashworth	Jan 6, 1987	\$23,456.78	Male
Jordan	Ashworth	Aug 31, 1989	\$34,567.89	Female

Table Selection Types: ☒ Row ☐ Column ☐ Cell

List Selection Modes: Multiple Interval Selections

Chú ý

(1): Điều này có lẽ chỉ đúng trong phiên bản java cũ mà tác giả dùng để viết ví dụ minh họa. Theo mình được biết bắt đầu từ JDK 1.5, lớp Boolean của chúng ta đã cài đặt interface Comparable và bằng chứng là khi chạy chương trình của chúng ta, cột Gender vẫn được sắp xếp bình thường như hình dưới đây:



First Name	Last Name	Date of Birth	Account Balance	Gender
Jordan	Ashworth	Aug 31, 1989	\$34,567.89	Female
Evelyn	Kirk	Jan 16, 1945	(\$456.70)	Female
Belle	Spyres	Aug 2, 1907	\$567.00	Female
Clay	Ashworth	Feb 20, 1962	\$12,345.67	Male
Jacob	Ashworth	Jan 6, 1987	\$23,456.78	Male

Table Selection Types: ☒ Row ☐ Column ☐ Cell

List Selection Modes: Multiple Interval Selections

First Name	Last Name	Date of Birth	Account Balance	Gender
Jacob	Ashworth	Jan 6, 1987	\$23,456.78	Male
Clay	Ashworth	Feb 20, 1962	\$12,345.67	Male
Belle	Spyres	Aug 2, 1907	\$567.00	Female
Evelyn	Kirk	Jan 16, 1945	(\$456.70)	Female
Jordan	Ashworth	Aug 31, 1989	\$34,567.89	Female

Table Selection Types
☒ Row
 ☐ Column
 ☐ Cell

List Selection Modes

Multiple Interval Selections

Các bạn cũng có thể tham khảo thêm tại [đây](#).

Như vậy sau bài viết này chúng ta đã nắm được cách để sắp xếp các hàng trong một bảng theo chiều tăng dần hoặc giảm dần của các giá trị trong một cột nào đó. Phần 9 sẽ trình bày cách thêm và xóa một dòng trong bảng.

PHẦN 9

Trong phần này chúng ta sẽ tìm hiểu về cách thêm và xóa dữ liệu ở trong một bảng. Để thay đổi dữ liệu trong bảng, chúng ta cần thay đổi TableModel và sau đó thông báo cho listener của nó (hay nói cách khác chính là JTable) rằng dữ liệu đã bị thay đổi.

Đoạn mã dưới đây là ví dụ minh họa cho một bảng có 1 cột. Có một ô chữ để cho phép chúng ta thêm những dòng mới vào trong bảng.

```
public class RowAdder extends JFrame{
    protected SimpleModel tableData;
    protected JTable table;
    protected JTextField textField;

    public static void main(String[] args) {
        RowAdder ra = new RowAdder();
        ra.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        ra.setSize(400, 300);
        ra.setVisible(true);
    }

    public RowAdder(){
        Container pane = getContentPane();
        pane.setLayout(new BorderLayout());
        tableData = new SimpleModel();
        table = new JTable(tableData);
        table.getColumnModel().getColumn(0).setPreferredWidth(300);
        JScrollPane jsp = new JScrollPane(table);
        pane.add(jsp, BorderLayout.CENTER);
        textField = new JTextField();
        textField.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent event) {
                addLineToTable();
            }
        });
    }
}
```

```
pane.add(textField, BorderLayout.SOUTH);
}

protected void addLineToTable() {
    tableData.addText(textField.getText());
    textField.setText("");
}

class SimpleModel extends AbstractTableModel {
    protected Vector textData = new Vector();

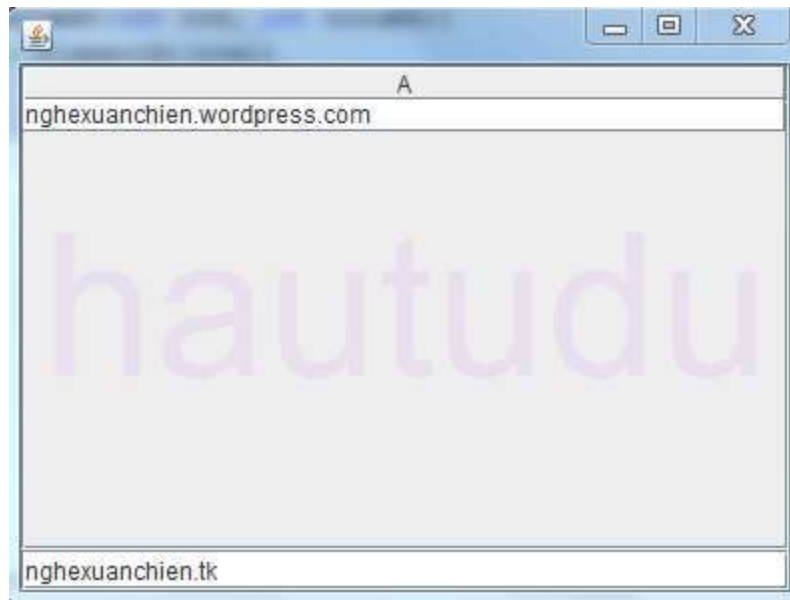
    public void addText(String text) {
        textData.addElement(text);
        fireTableDataChanged();
    }

    public int getRowCount() {
        return textData.size();
    }

    public int getColumnCount() {
        return 1;
    }

    public Object getValueAt(int row, int column) {
        return textData.elementAt(row);
    }
}
}
```

Khi chạy chương trình, chúng ta nhập chữ vào trong ô chữ sau đó nhấn Enter. Một dòng mới trong sẽ được thêm vào JTable như hình dưới đây:



Khi nhấn phím Enter vào ô chữ, một sự kiện sẽ được sinh ra và xử lý sự kiện đó sẽ gọi phương thức **fireTableDataChanged()** để cập nhật lại dữ liệu trong bảng. Tuy nhiên phương thức này chỉ được cài đặt trước trong lớp `AbstractTableModel`, còn nếu cái model mà không phải kế thừa từ lớp `AbstractTableModel` thì chúng ta sẽ không có phương thức này để dùng. Lúc đó sẽ phải tự tạo ra một phương thức trong model giống như là phương thức `fireTableDataChanged()` của lớp `AbstractTableModel`. Ví dụ dưới đây minh họa trong trường hợp model của chúng ta cài đặt interface `TableModel` thì sẽ làm như sau:

```
public class RowAdder extends JFrame {
    protected SimpleModel tableData;
    protected JTable table;
    protected JTextField textField;

    public static void main(String[] args) {
        RowAdder ra = new RowAdder();
        ra.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        ra.setSize(400, 300);
        ra.setVisible(true);
    }
}
```

```

public RowAdder(){
    Container pane = getContentPane();
    pane.setLayout(new BorderLayout());
    tableData = new SimpleModel();
    table = new JTable(tableData);
    table.getColumnModel().getColumn(0).setPreferredWidth(300);
    JScrollPane jsp = new JScrollPane(table);
    pane.add(jsp, BorderLayout.CENTER);
    textField = new JTextField();
    textField.addActionListener(new ActionListener() {

        public void actionPerformed(ActionEvent event) {
            addLineToTable();
        }
    });
    pane.add(textField, BorderLayout.SOUTH);
}

protected void addLineToTable(){
    tableData.addText(textField.getText());
    textField.setText("");
}

class SimpleModel implements TableModel{
    protected Vector textData = new Vector();
    protected EventListenerList listenerList = new EventListenerList();

    public void addText(String text){
        textData.addElement(text);
        notifyListenersOfDataChange();
    }
}

```

```

public void notifyListenersOfDataChange(){
    TableModelEvent event= new TableModelEvent(this);
    Object[] listeners = listenerList.getListenerList();
    for(int i = 0; i< listeners.length; i++){
        if(listeners[i] == TableModelListener.class){
            TableModelListener listener = (TableModelListener) (listeners[i+1]);
            listener.tableChanged(event);
        }
    }
}

public int getRowCount(){
    return textData.size();
}

public int getColumnCount(){
    return 1;
}

public Object getValueAt(int row, int column){
    return textData.elementAt(row);
}

public String getColumnName(int columnIndex) {
    return "A";
}

public Class getColumnClass(int columnIndex) {
    return String.class;
}

public boolean isCellEditable(int rowIndex, int columnIndex) {

```

```

        return false;
    }

    public void setValueAt(Object aValue, int rowIndex, int columnIndex) {
        textData.setElementAt(aValue, rowIndex);
    }

    public void addTableModelListener(TableModelListener l) {
        listenerList.add(TableModelListener.class, l);
    }

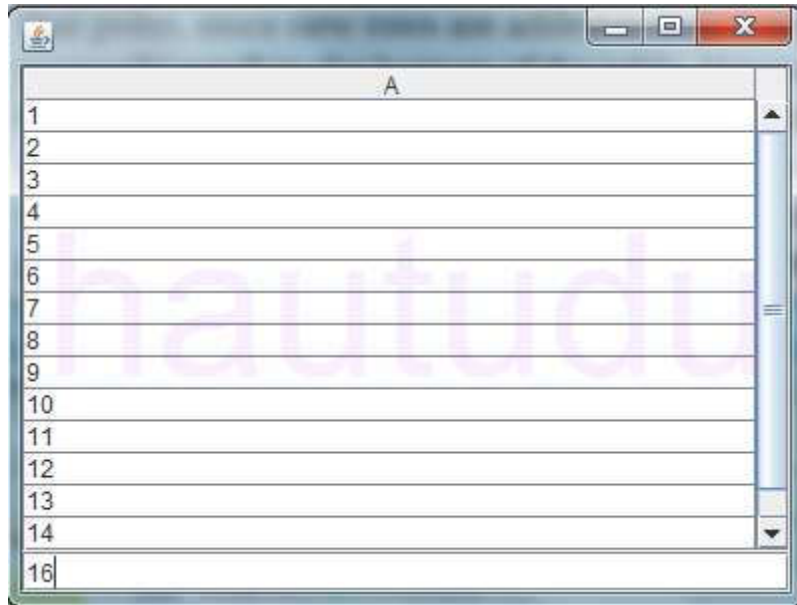
    public void removeTableModelListener(TableModelListener l) {
        listenerList.remove(TableModelListener.class, l);
    }
}

```

Trong đoạn mã trên, chúng ta tạo ra phương thức **notifyListenersOfDataChange()** và phương thức này tương đương với phương thức **fireTableDataChanged()** khi dùng AbstractTableModel. Bản chất vẫn là gọi phương thức **tableChanged()** của các listener và truyền vào đó một đối tượng của **TableModelEvent**. Các listener sẽ được đăng kí với một model qua phương thức **addTableModelListener()** của model đó. Listener trong trường hợp này chính là cái JTable đang tham chiếu đến cái model SimpleModel.

Hiện thị một dòng trong bảng

Trong ví dụ bên trên, khi mà chúng ta nhập chữ vào ô chữ rồi ấn phím Enter, một dòng mới sẽ được thêm vào bảng. Khi một vài dòng đầu được thêm vào, chúng ta sẽ nhìn thấy chúng ngay lập tức. Tuy nhiên, khi số dòng nhiều vượt quá kích thước thì sẽ dẫn đến sự xuất hiện của thanh cuộn dọc. Và đến đây, vấn đề chính là không thể nhìn thấy được dòng vừa được thêm vào bảng trừ khi chúng ta phải cuộn thanh cuộn xuống dưới. Hình dưới đây minh họa cho việc khi mà nhập quá 14 dòng vào bảng, nếu không cuộn thanh cuộn xuống dưới, thì từ dòng 15 trở xuống, chúng ta sẽ không nhìn thấy được.



Như vậy, chúng ta phải tìm cách nào đó để cho thanh cuộn tự cuộn xuống khi mà một dòng mới được thêm vào bảng. Có làm điều này bằng cách lấy ra đối tượng JViewport của cửa sổ cuộn và sau đó đặt lại vị trí cho nó sao cho dòng cuối cùng của bảng luôn được hiện lên trong cửa sổ cuộn. Chúng ta sửa lớp RowAdder như sau:

```
public class RowAdder extends JFrame {  
    protected SimpleModel tableData;  
    protected JTable table;  
    protected JTextField textField;  
  
    public static void main(String[] args) {  
        RowAdder ra = new RowAdder();  
        ra.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        ra.setSize(400, 300);  
        ra.setVisible(true);  
    }  
  
    public RowAdder() {  
        Container pane = getContentPane();  
        pane.setLayout(new BorderLayout());  
        tableData = new SimpleModel();
```

```

table = new JTable(tableData);
table.getColumnModel().getColumn(0).setPreferredWidth(300);
table.addComponentListener(new TableScroller());
JScrollPane jsp = new JScrollPane(table);
pane.add(jsp, BorderLayout.CENTER);
textField = new JTextField();
textField.addActionListener(new ActionListener() {

    public void actionPerformed(ActionEvent event) {
        addLineToTable();
    }
});
pane.add(textField, BorderLayout.SOUTH);
}

protected void addLineToTable(){
    tableData.addText(textField.getText());
    textField.setText("");
}

class SimpleModel implements TableModel{
    [...]
}

class TableScroller extends ComponentAdapter{
    public void componentResized(ComponentEvent event){
        int lastRow = tableData.getRowCount() -1;
        int cellTop = table.getCellRect(lastRow, 0, true).y;
        JScrollPane jsp = (JScrollPane)
SwingUtilities.getAncestorOfClass(JScrollPane.class, table);
        JViewport jvp = jsp.getViewport();
        int portHeight = jvp.getSize().height;

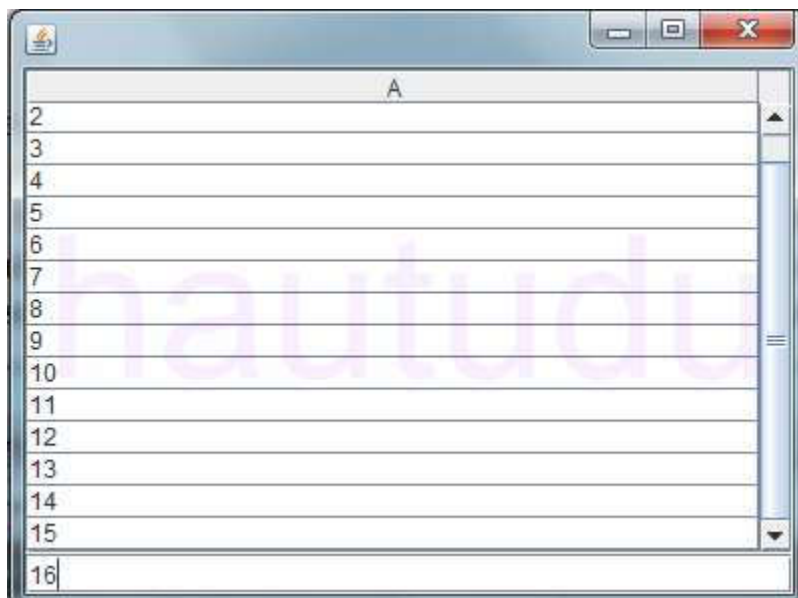
```

```

        int position = cellTop - (portHeight - table.getRowHeight() -
table.getRowMargin());
        if(position >= 0){
            jvp.setViewPosition(new Point(0, position));
        }
    }
}
}

```

Phương thức **componentResized()** lấy về kích thước và tọa độ của một dòng trong bảng bằng việc gọi phương thức **getCellRect()**. Sau đó nó sử dụng vị trí thẳng đứng của dòng, kích thước của viewport, và chiều cao của dòng để điều chỉnh vị trí nhìn của viewport sao cho chúng ta có thể luôn nhìn thấy được dòng cuối cùng của bảng. Sau khi chạy chương trình, bây giờ chúng ta thêm dòng đến đâu, thanh cuộn sẽ tự động cuộn xuống tới đó như hình dưới đây:



Như vậy, trong phần này chúng ta đã biết cách để thêm một dòng mới vào bảng như thế nào. Thêm vào đó, chúng ta cũng biết cách để hiển thị một dòng tại đúng vị trí mà chúng ta mong muốn. Loạt bài “Cách sử dụng JTable của Swing trong java” sẽ dừng ở đây với hi vọng cung cấp cho bạn đọc những kiến thức cơ bản về cách sử dụng JTable – một thành phần rất hay được sử dụng trong các ứng dụng viết bằng java.