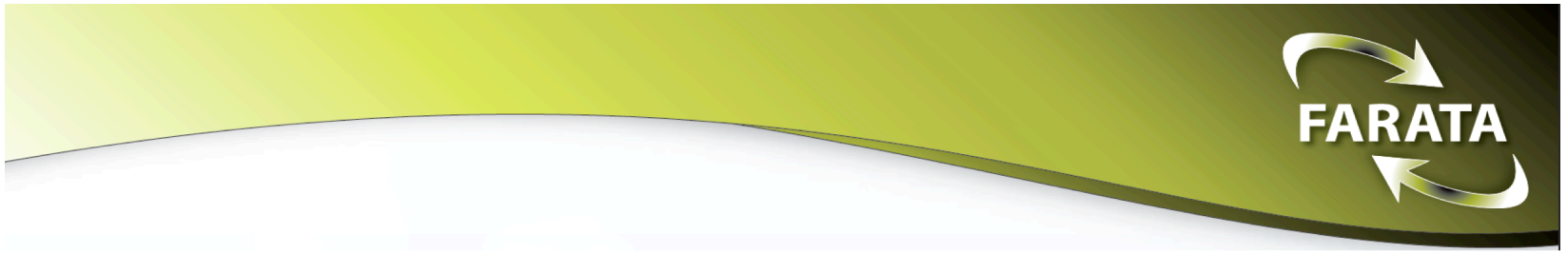


Java Programming
Unit 13

Working with Swing JTable
Annotations.
Reflection

Swing JTable

JTable and the MVC Paradigm

- The Swing class `JTable` is a UI component for displaying tabular spreadsheet-like data.

- The data is represented as rows and columns, which make `JTable` a good choice for displaying records from RDBMS tables.

M, V, and C

`JTable` implements *Model-View-Controller (***MVC***)* design pattern - presentation components (the view) are separated from components that store data (the model).

`JTable` displays (**V**) the data (**M**) stored in a different Java class that implements the `TableModel` interface.

Any class can be a controller (**C**) if it initiate actions to move the data from model to view or vice versa. E.g. a `JButton` click initiates the population of the `TableModel` from the database and displays the data in `JTable`.

The Model

- Swing classes `DefaultTableModel` and `AbstractTableModel` implement the `TableModel` interface and have methods to notify a `JTable` when the data is changing.

- A programmer can create a model as a subclass of `AbstractTableModel` to store the data in some collection, e.g. `ArrayList`.

- The UI class that creates `JTable` defines one or more listeners to be notified of any changes in the table's data.

A Window with a JTable

```
public class MyFrame  extends JFrame implements TableModelListener{

 private MyTableModel myTableModel;
 private JTable myTable;

 MyFrame (String winTitle){
 super(winTitle);

 myTableModel = new MyTableModel();
 myTable = new JTable(myTableModel );

 //Add the JTable to frame and enable scrolling
 add(new JScrollPane( myTable));

 // Register an event listener
  myTableModel.addTableModelListener(this);
}
public void tableChanged(TableModelEvent e) {
 // Code to process data changes goes here
}
public static void main(String args[]){
 MyFrame myFrame = new MyFrame( "My Test Window" );

 myFrame.pack();
 myFrame.setVisible( true );
}

 class MyTableModel extends AbstractTableModel {
   // The data for JTable should be here
  }
}
```

This is an example of a model with hard-coded data:

```
ArrayList<Order> myData = new ArrayList<Order>();

myData.add(new Order(1,"IBM", 100, 135.5f));
myData.add(new Order(2,"AAPL", 300, 290.12f));
myData.add(new Order(3,"MOT", 2000, 8.32f));
myData.add(new Order(4,"ORCL", 500, 27.8f));
```

Mandatory Callbacks of `JTableModel`

The class that implements the `TableModel` interface and feeds the data to `Jtable` must include at least three callbacks:

```
getColumnCount()  // get the number of columns in the JTable

getRowCount()     // get the number of rows in the JTable

getValueAt(int row, int col)// returns the  Object  with
                            // the value in the cell
```

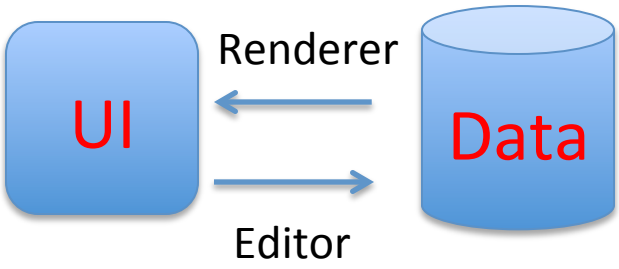
Renderers and Editors

The data transfer from the table model to `JTable` is performed by *cell renderers*.

Default cell renderer extends `JLabel`, hence all the data are displayed as text.

But you can create a custom cell renderer.

When the user is modifying the content of the cell, the *cell editor* is engaged.




```
//Assign custom cell renderer to the Price column
 // Get the reference to the fourth column - Price
 TableColumn column = myTable.getColumnModel().getColumn(3);

 // Create a new cell renderer as an anonymous inner
 // class and assign it to the column price
 column.setCellRenderer(
      new DefaultTableCellRenderer(){
       public Component  getTableCellRendererComponent(
         JTable table, Object value, boolean isSelected,
                boolean hasFocus, int row, int col) {

       JLabel label = (JLabel) super.getTableCellRendererComponent(
           table, value, isSelected, hasFocus, row, col);

     // right-align the price value
     label.setHorizontalAlignment(JLabel.RIGHT);

     // display stocks that cost more than $100 in red
     if (((Float) value)>100){
       label.setForeground(Color.RED);
     } else{
         label.setForeground(Color.BLACK);
     }
     return label;

    } // end of getTableCellRendererComponent
   }  // end of new DefaultTableCellRenderer
 );    // end of setCellRenderer(...)
```

Walkthrough 1

- Import the code sample from Lesson 23 into Eclipse.
- Run the program MyFrame and review the output.
- Run the program MyFrameWithCustomRenderer, which uses a custom renderer and review the output.

Financial Dashboard

Annotations

Annotations

- *Metadata is the data about your data, a document, or any other artifact.*

- Program's metadata is the data about your code.

- Any Java class contains its metadata, and a class can ask another class, *"What methods do you have?"*

- You can declare custom *annotations* and define your own processing rules that will route the execution of your program, produce configuration files, additional code, deployment descriptors etc.

- When you create annotations, you also need to *create or use an annotation processor* to get the expected output.

Predefined Java Annotations

- There are about a dozen of predefined annotations in Java SE, the packages java.lang, java.lang.annotation, and javax.annotation.

- Some of these annotations are used by the compiler: `@Override, @SuppressWarning, @Deprecated, @Target, @Retention, @Documented, @Inherited, @FunctionalInterface`.

- Some are used by the Java SE run-time or third-party run times and indicate methods that have to be invoked in a certain order (`@PostConstruct, @PreDestroy`), or mark code that was generated by third-party tools (`@Generated`).

- In Java EE annotations are being used everywhere.

@Override

```
public class NJTax extends Tax {


 @override
 public double calcTax() {
    double  stateTax=0;
    if (grossIncome < 30000) {

       stateTax=grossIncome*0.05;

    } else{
       stateTax= grossIncome*0.06;
    }

       return stateTax - 500;
    }
}
```

```
class Tax{
    double grossIncome;
    String state;
    int dependents;

    public double calcTax() {
        double  stateTax=0;
        if (grossIncome < 30000) {
          stateTax=grossIncome*0.05;
        }
        else{
          stateTax= grossIncome*0.06;
        }
          return stateTax;
  }
}
```

Try changing `calcTax()` in `NJTax`:

```
@Override public double calcTax(String something)
```

Compiler gives an error:

The method `calcTax(String)` of type `NJTax` must override or implement a supertype method

Custom Annotations

Java has a mechanism for creation of your own annotations and annotation processors.

For example, it's possible to can to create an annotation that will allow other programmers to mark class methods with an SQL statement to be executed during the run time.

To create custom annotations:

a) Declare your own annotation to be used by other Java classes.

b) Write the annotation processor that will read these Java classes, identify and parse annotations, and process them accordingly.

A Sample Custom Annotation

A custom annotation →

```
@Target({ ElementType.METHOD,
ElementType.CONSTRUCTOR })
@Retention(RetentionPolicy.SOURCE)
public @interface MyJDBCExecutor {
  String value();
}
```

An annotation processor is needed here

```
class HRBrowser{

  @MyJDBCExecutor (value="Select * from Employee")
  public List getEmployees(){

     // add calls to some JDBC executing engine here
   }

}
```

An annotation with three arguments

A custom annotation with 3 args →

Annotations with the SOURCE retention policy can be processed by the Annotation Processing Tool (APT), but it's **deprecated** as of Java 7.

Now javac directly supports annotations, see http://bit.ly/13Bw8up

For annotations with `RUNTIME` retention policy you should know how to write an annotation processor, which has to "extract" the values from the annotations during run time and invoke appropriate code.

```
import java.lang.annotation.*;

@Target({ ElementType.METHOD})
@Retention(RetentionPolicy.SOURCE)
public @interface MyJDBCExecutor {
        String sqlStatement();
        boolean transactionRequired() default false;
        boolean notifyOnUpdates() default false;
}
```

```
class HRBrowser{

 @MyJDBCExecutor (sqlStatement="Select * from Employee")
 public List<Employee> getEmployees(){
    // The code to get the the data from DBMS goes here,

    // result set goes in ArrayList, which is returned to the

    // caller of the method getEmployees()

    …
    return myEmployeeList;
 }
}
```

Reflection

Java Reflection

During the run time you can query about the internals of a Java class (its methods, constructors, and fields) and invoke the discovered methods or access public member variables.

A special class called `Class` represents classes of your program. In particular, it can load a class in memory, and then you can explore the content of this class by using classes from the package java.lang.reflect.

Read more about the class `Class` at http://docs.oracle.com/javase/8/docs/api/java/lang/Class.html

Reflection Sample

```
abstract public class Person {
  abstract public void raiseSalary();
}
```

```
public class Employee extends Person{
 public void raiseSalary() {
   System.out.println(
         "Raising salary for Employee...");
 }
}
```

The ReflectionSample class loads the class Employee, prints its method signatures, and finds its superclass and its methods.

The process of querying an object about its content during run time is called *introspection*.

```
import java.lang.reflect.*;
public class ReflectionSample {
 public static void main(String args[]) {
   try {
     Class c = Class.forName("Employee"); // load class Employee
     Method methods[] = c.getDeclaredMethods();
     System.out.println("The  Employee methods:");

     for (int i = 0; i < methods.length; i++){
         System.out.println("*** Method Signature:" +
                            methods[i].toString());
     }

     Class superClass = c.getSuperclass();
     System.out.println("The name of the superclass is "
                        + superClass.getName());

     Method superMethods[] =superClass.getDeclaredMethods();
     System.out.println("The superclass has:");

     for (int i = 0; i < superMethods.length; i++){
         System.out.println("*** Method Signature:" +
                        superMethods[i].toString());
         System.out.println("      Return type: " +
           superMethods[i].getReturnType().getName());
     }
   } catch (Exception e) {
       e.printStackTrace();
   }
 }
}
```

Walkthrough 2

- Download and import into Eclipse the source code for Lesson 24 in the textbook.
- Review the code of ReflectionSample and run the program.

- Observe the output of the program.

Runtime Annotation Processing

- Our class `MyJDBCAnnotationProcessor` will load the class HRBrowser, introspect its content, find the annotations and their values, and process them accordingly.

```
import java.lang.annotation.*;
@Inherited
@Documented
@Target({ ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface MyJDBCExecutor {
        //String value();
          String sqlStatement();
          boolean transactionRequired() default false;
          boolean notifyOnUpdates() default false;

}
```

We will reuse the annotation MyJDBCExecutor, but will change the retention policy to RUNTIME

Runtime Annotation Processing (cont.)

We'll review the annotation processor class called `MyJDBCAnnotationProcessor`, and the class `HRBrowser` will serve as a command line argument to that processor:

```
c:/>java MyJDBCAnnotationProcessor HRBrowser
```

The class `MyJDBCAnnotationProcessor` has to load the class `HRBrowser`, introspect its content, find the annotations and their values, and process them accordingly. But we'll just do the annotation-discovery part.

Walkthrough 3

- Run the program MyJDBCAnnotationProcessor

- You'll get an error. Fix it and run the program again. It should print the following:

Method: getEmployees. Parameters of MyJDBCGenerator are: sqlStatement=Select * from Employee, notifyOnUpdates=false, transactionRequired=false

Method: updateData. Parameters of MyJDBCGenerator are: sqlStatement=Update Employee set bonus=1000, notifyOnUpdates=true, transactionRequired=true

- Review the code of the program.

Homework

1. Do the assignment from the Try It sections of Lesson 23 and 24.

2. Get familiar with one of the components libraries like Apache Commons (http://commons.apache.org) or Guava (https://code.google.com/p/guava-libraries). Pick any component and write a little program that uses it.

Extra Challenge

- Get familiar with the usage of regular expressions. Watch this video: https://www.youtube.com/watch?v=EkluES9Rvak#at=15

- Create a simple application with UI, where the user has to enter something in the field (e.g. email, phone number) and when the focus leaves this field validate the input using a regular expression.

Additional Reading

1. Google Annotations Gallery: http://code.google.com/p/gag/
2. The @Override annotation: http://bit.ly/qzyCgk
3. Code generation using annotations:
   http://deors.wordpress.com/2011/09/26/annotation-types/
4. How to compare objects for equality:
   http://javarevisited.blogspot.com/2011/06/comparator-and-comparable-in-java.html
5. Top 10 mistakes Java programmers make:
   http://java.dzone.com/articles/top-10-lists-common-java
6. Get familiar with the GUI library jGoodies: http://www.jgoodies.com/