vogella.com

Introduction to the Eclipse JFace Table API - Tutorial

Lars Vogel, Clemens Muessener (c) 2008, 2017 vogella GmbH
Version 3.2,
06.07.2016
Eclipse JFace Table. This tutorial explains the usage of Eclipse JFace TableViewer.

1. JFace Table Viewer
1.1. Using the JFace TableViewer
1.2. Content provider for JFace tables
1.3. Columns and label provider
1.4. Reflect data changes in the viewer
1.5. Selection change listener
1.6. Selecting elements of the Table
2. Prerequisites
3. Tutorial: JFace Table Viewer
3.1. Overview of the example
3.2. Project creation and data model
3.3. Define the viewer
3.4. Run
4. Learn more about Eclipse RCP development
5. Download
6. JFace Resources
7. vogella training and consulting support
Appendix A: Copyright, License and Source code

1. JFace Table Viewer

1.1. Using the JFace TableViewer

You can use the TableViewer class to create tables using the JFace framework. The SWT Table widget is wrapped into the TableViewer. The table widget can still be accessed to set its properties.

```
// define the TableViewer
viewer = new TableViewer(parent, SWT.MULTI | SWT.H_SCROLL
        | SWT.V_SCROLL | SWT.FULL_SELECTION | SWT.BORDER);

// create the columns
// not yet implemented
createColumns(viewer);

// make lines and header visible
final Table table = viewer.getTable();
table.setHeaderVisible(true);
table.setLinesVisible(true);
```

1.2. Content provider for JFace tables

As with other JFace viewers a content provider supplies the data which should be displayed in the TableViewer.

Eclipse provides an implementation of this interface via the ArrayContentProvider class. The ArrayContentProvider class supports Arrays or Collections as input, containing the domain data. You can implement your own content provider for a table by implementing the interface IStructuredContentProvider from the org.eclipse.jface.viewers package.

The getElements() method of the content provider is used to translate the input of the viewer into an array of elements. Once the setInput() method on the viewer is called, it uses the content provider to convert it. This is the reason why the content provider must be set before the setInput() method is called.

Each object in the Array returned by the content provider is displayed as individual element by the viewer. In case of the table viewer each object is displayed in an individual row.

The usage of the content provider is demonstrated with the following code snippet.

```
// this code is placed after the definition of
// the viewer

// set the content provider
viewer.setContentProvider(ArrayContentProvider.getInstance());

// provide the input to the viewer
// setInput() calls getElements() on the
// content provider instance
viewer.setInput(someData...);
```

1.3. Columns and label provider

Columns for a JFace TableViewer object are defined by creating instances of the TableViewerColumn class.

Each TableViewerColumn object needs to get a label provider assigned to it via the setLabelProvider() method. The label provider defines which data is displayed in the column. The label provider for a table viewer column is called per row and gets the corresponding object as input. It uses this input to determine which data is displayed in the column for this row.

Typically the label provider returns a String, but more complex implementations are possible.

The setLabelProvider() method on the TableViewerColumn expects an instance of the abstract CellLabelProvider class. A default implementation of this class is provided by the ColumnLabelProvider class. Its usage is demonstrated in the following code snippet.

```
// create a column for the first name
TableViewerColumn colFirstName = new TableViewerColumn(viewer, SWT.NONE);
colFirstName.getColumn().setWidth(200);
colFirstName.getColumn().setText("Firstname");
colFirstName.setLabelProvider(new ColumnLabelProvider() {
   @Override
   public String getText(Object element) {
      Person p = (Person) element;
      return p.getFirstName();
   }
});

// create more text columns if required...

// create column for married property of Person
// uses getImage() method instead of getText()
// CHECKED and UNCHECK are fields of type Image

TableViewerColumn colMarried = new TableViewerColumn(viewer, SWT.NONE);
colMarried.getColumn().setWidth(200);
colMarried.getColumn().setText("Married");
colMarried.setLabelProvider(new ColumnLabelProvider() {

   private ResourceManager resourceManager = new
LocalResourceManager(JFaceResources.getResources());

   @Override
   public String getText(Object element) {
      return null;  // no string representation, we only want to display the image
   }

   @Override
   public Image getImage(Object element) {
      if (((Person) element).isMarried()) {
         return resourceManager.createImage(CHECKED);
      }
      return resourceManager.createImage(UNCHECKED);
   }

   @Override
   public void dispose() {
      super.dispose();
```

```
        resourceManager.dispose();
    }
});
```
The above code uses two fields which contain Image instances. These fields could for example be initialized via the following code. Using the classes in this code requires a dependency to the org.eclipse.core.runtime plug-in.

```
// fields for your class
// assumes that you have these two icons
// in the "icons" folder
private final ImageDescriptor CHECKED = getImageDescriptor("checked.gif");
private final ImageDescriptor UNCHECKED = getImageDescriptor("unchecked.gif");


// more code...


private static ImageDescriptor getImageDescriptor(String file) {
    // assume that the current class is called View.java
    Bundle bundle = FrameworkUtil.getBundle(View.class);
    URL url = FileLocator.find(bundle, new Path("icons/" + file), null);
    return ImageDescriptor.createFromURL(url);
}
```
1.4. Reflect data changes in the viewer
To reflect data changes in the data model that is displayed by the viewer, you can call the viewer.refresh() method. This method updates the viewer based on the data which is assigned to it.

To change the data which is displayed use the viewer.setInput() method.

1.5. Selection change listener
Via the addSelectionChangedListener method you can add a listener to a viewer. This listener is an implementation of the ISelectionChangedListener interface. The following code shows an example that gets the selected element of the viewer.

```
viewer.addSelectionChangedListener(new ISelectionChangedListener() {
    @Override
    public void selectionChanged(SelectionChangedEvent event) {
        IStructuredSelection selection = viewer.getStructuredSelection();
        Object firstElement = selection.getFirstElement();
        // do something with it
    }
});
```

1.6. Selecting elements of the Table
There are many different ways how an element in a table can be selected.

Either the low level SWT Table API can be use:

```
viewer.getTable().select(int index);
viewer.getTable().select(int[] indices);
viewer.getTable().select(int start, int end);
viewer.getTable().selectAll();
```

Or the JFace TableViewer API:

```
StructuredSelection structuredSelection = new StructuredSelection(listOfPersons);
viewer.setSelection(structuredSelection);
```

or

```
// Second param is used to reveal (make visible) the selection in the viewer
viewer.setSelection(structuredSelection, true);
```

In most of the cases it is desired to select certain objects and therefore using the JFace API is more common. But if the indices are known or simply all items should be selected the low level SWT API is sufficient.

2. Prerequisites
The following provides an example how to build a table with the JFace Viewer framework.

It assume that you are familiar with creating Eclipse RCP applications or Eclipse Plug-ins.

Please see Introduction to JFace for an introduction to the concepts behind this example.

3. Tutorial: JFace Table Viewer
3.1. Overview of the example
We will build an Eclipse RCP application which displays data of persons in a JFace table. Each person is displayed in one individual row. This tutorial the basic setup of a JFace Table.

The final application will look like this.

jfacetable10
3.2. Project creation and data model
Create a new RCP Project de.vogella.jface.tableviewer using the "RCP application with a view" as a template. Create a package "de.vogella.jface.tableviewer.model" and the following class "Person".

```
}
```

```
package de.vogella.jface.tableviewer.model;

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;

public class Person {
    private String firstName;
    private String lastName;
    private boolean married;
    private String gender;
    private Integer age;
    private PropertyChangeSupport propertyChangeSupport = new PropertyChangeSupport(
            this);

    public Person() {
    }

    public Person(String firstName, String lastName, String gender,
            boolean married) {
        super();
        this.firstName = firstName;
        this.lastName = lastName;
        this.gender = gender;
        this.married = married;
    }

    public void addPropertyChangeListener(String propertyName,
            PropertyChangeListener listener) {
        propertyChangeSupport.addPropertyChangeListener(propertyName, listener);
    }

    public void removePropertyChangeListener(PropertyChangeListener listener) {
        propertyChangeSupport.removePropertyChangeListener(listener);
    }

    public String getFirstName() {
        return firstName;
    }

    public String getGender() {
        return gender;
    }
    public String getLastName() {
        return lastName;
    }

    public boolean isMarried() {
        return married;
    }

    public void setFirstName(String firstName) {
        propertyChangeSupport.firePropertyChange("firstName", this.firstName,
                this.firstName = firstName);
    }

    public void setGender(String gender) {
        propertyChangeSupport.firePropertyChange("gender", this.gender,
                this.gender = gender);
    }

    public void setLastName(String lastName) {
        propertyChangeSupport.firePropertyChange("lastName", this.lastName,
                this.lastName = lastName);
    }

    public void setMarried(boolean isMarried) {
        propertyChangeSupport.firePropertyChange("married", this.married,
                this.married = isMarried);
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        propertyChangeSupport.firePropertyChange("age", this.age,
                this.age = age);
    }

    @Override
    public String toString() {
        return firstName + " " + lastName;
    }

}
```

The Person class represents the data model for this example. It has also PropertyChange support, which is not necessary for this example but is nice if you would later extend this example with Eclipse Data Binding support.

Create the ModelProvider class which is a in-memory representation of your data. This class is defined as a Singleton.

```
package de.vogella.jface.tableviewer.model;

import java.util.ArrayList;
import java.util.List;

public enum ModelProvider {
  INSTANCE;

  private List<Person> persons;

  private ModelProvider() {
    persons = new ArrayList<Person>();
    // Image here some fancy database access to read the persons and to
    // put them into the model
    persons.add(new Person("Rainer", "Zufall", "male", true));
    persons.add(new Person("Reiner", "Babbel", "male", true));
    persons.add(new Person("Marie", "Dortmund", "female", false));
    persons.add(new Person("Holger", "Adams", "male", true));
    persons.add(new Person("Juliane", "Adams", "female", true));
  }

  public List<Person> getPersons() {
    return persons;
  }

}
```

3.3. Define the viewer
Change the View

```
package de.vogella.jface.tableviewer;

import org.eclipse.jface.viewers.ArrayContentProvider;
import org.eclipse.jface.viewers.ColumnLabelProvider;
import org.eclipse.jface.viewers.TableViewer;
import org.eclipse.jface.viewers.TableViewerColumn;
import org.eclipse.swt.SWT;
import org.eclipse.swt.graphics.Image;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Table;
import org.eclipse.swt.widgets.TableColumn;
import org.eclipse.swt.widgets.Text;
import org.eclipse.ui.part.ViewPart;

import de.vogella.jface.tableviewer.model.ModelProvider;
import de.vogella.jface.tableviewer.model.Person;

public class View extends ViewPart {
   public static final String ID = "de.vogella.jface.tableviewer.view";

   private TableViewer viewer;
   // static fields to hold the images
   private static final ImageDescriptor CHECKED = getImageDescriptor("checked.gif");
   private static final ImageDescriptor UNCHECKED = getImageDescriptor("unchecked.gif");

   public void createPartControl(Composite parent) {
      GridLayout layout = new GridLayout(2, false);
      parent.setLayout(layout);
      Label searchLabel = new Label(parent, SWT.NONE);
      searchLabel.setText("Search: ");
      final Text searchText = new Text(parent, SWT.BORDER | SWT.SEARCH);
      searchText.setLayoutData(new GridData(GridData.GRAB_HORIZONTAL |
GridData.HORIZONTAL_ALIGN_FILL));
      createViewer(parent);
   }

   private void createViewer(Composite parent) {
      viewer = new TableViewer(parent, SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL |
SWT.FULL_SELECTION | SWT.BORDER);
      createColumns(parent, viewer);
      final Table table = viewer.getTable();
      table.setHeaderVisible(true);
      table.setLinesVisible(true);

      viewer.setContentProvider(new ArrayContentProvider());
      // get the content for the viewer, setInput will call getElements in the
      // contentProvider
```

```
    viewer.setInput(ModelProvider.INSTANCE.getPersons());
    // make the selection available to other views
    getSite().setSelectionProvider(viewer);
    // set the sorter for the table

    // define layout for the viewer
    GridData gridData = new GridData();
    gridData.verticalAlignment = GridData.FILL;
    gridData.horizontalSpan = 2;
    gridData.grabExcessHorizontalSpace = true;
    gridData.grabExcessVerticalSpace = true;
    gridData.horizontalAlignment = GridData.FILL;
    viewer.getControl().setLayoutData(gridData);
}

public TableViewer getViewer() {
    return viewer;
}

// create the columns for the table
private void createColumns(final Composite parent, final TableViewer viewer) {
    String[] titles = { "First name", "Last name", "Gender", "Married" };
    int[] bounds = { 100, 100, 100, 100 };

    // first column is for the first name
    TableViewerColumn col = createTableViewerColumn(titles[0], bounds[0], 0);
    col.setLabelProvider(new ColumnLabelProvider() {
        @Override
        public String getText(Object element) {
            Person p = (Person) element;
            return p.getFirstName();
        }
    });

    // second column is for the last name
    col = createTableViewerColumn(titles[1], bounds[1], 1);
    col.setLabelProvider(new ColumnLabelProvider() {
        @Override
        public String getText(Object element) {
            Person p = (Person) element;
            return p.getLastName();
        }
    });
```

```
        // now the gender
        col = createTableViewerColumn(titles[2], bounds[2], 2);
        col.setLabelProvider(new ColumnLabelProvider() {
            @Override
            public String getText(Object element) {
                Person p = (Person) element;
                return p.getGender();
            }
        });

        // now the status married
        col = createTableViewerColumn(titles[3], bounds[3], 3);
        col.setLabelProvider(new ColumnLabelProvider() {
            @Override
            public String getText(Object element) {
                return null;
            }

            @Override
            public Image getImage(Object element) {
                if (((Person) element).isMarried()) {
                    return CHECKED;
                } else {
                    return UNCHECKED;
                }
            }
        });

    }

    private TableViewerColumn createTableViewerColumn(String title, int bound, final int
colNumber) {
        final TableViewerColumn viewerColumn = new TableViewerColumn(viewer, SWT.NONE);
        final TableColumn column = viewerColumn.getColumn();
        column.setText(title);
        column.setWidth(bound);
        column.setResizable(true);
        column.setMoveable(true);
        return viewerColumn;
    }

    public void setFocus() {
        viewer.getControl().setFocus();
    }
}
```
The createColumns method creates the table columns, headers, sets the size of the columns and makes the columns re-sizable.