Adding Custom Column Property Accessor:

Create a Class called ColumnPropertyAccessor which extends `IColumnPropertyAccessor`.

```
// IColumnPropertyAccessor<Contact> columnPropertyAccessor = new
// ReflectiveColumnPropertyAccessor<Contact>(
// propertyNames);
ColumnPropertyAccessor columnPropertyAccessor = new ColumnPropertyAccessor();
IDataProvider bodyDataProvider = new ListDataProvider<Contact>(addressBook.getContacts(),
        columnPropertyAccessor);
```

ColumnPropertyAccessor Class.

Inside the class we pass propertyNames as list as follows

```
private static final List<String> propertyNames = Arrays.asList("name", "address",
"contact");
```

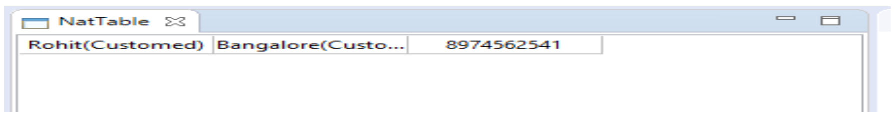
This propertyNames should be used inside the methods such as getColumnProperty()and getColumnIndex().

```
    @Override
    public Object getDataValue(Contact contact, int columnIndex) {
        switch (columnIndex) {
        case 0:
            return contact.getName()+ "(Customed)";
        case 1:
            return contact.getAddress()+ "(Customed)" ;
        case 2:
            return contact.getContact();
        }
        return null;
    }

    @Override
    public void setDataValue(Contact contact, int columnIndex, Object newValue) {
        switch (columnIndex) {
        case 0:
            String name = String.valueOf(newValue);
            contact.setName(name);
            break;
        case 1:
            String address = String.valueOf(newValue);
            contact.setAddress(address);
            break;
        case 2:
            String contactNumber = String.valueOf(newValue);
            contact.setContact(contactNumber);
            break;
        }
    }
```

In getColumnCount() the number of columns we used in our nattable.

In getColumnProperty()and getColumnIndex() methods we just pass the propertynames.

Run the Application :



Rohit(Customed)	Bangalore(Custo...	8974562541

Adding Column and Row Header :

Setting up the column header region

Since the column header has a dependence on the body layer and hence inherits features from it. All it needs to do in most cases is to have a data provider. This data provider will supply data for the column labels.

Setting up the row header layer

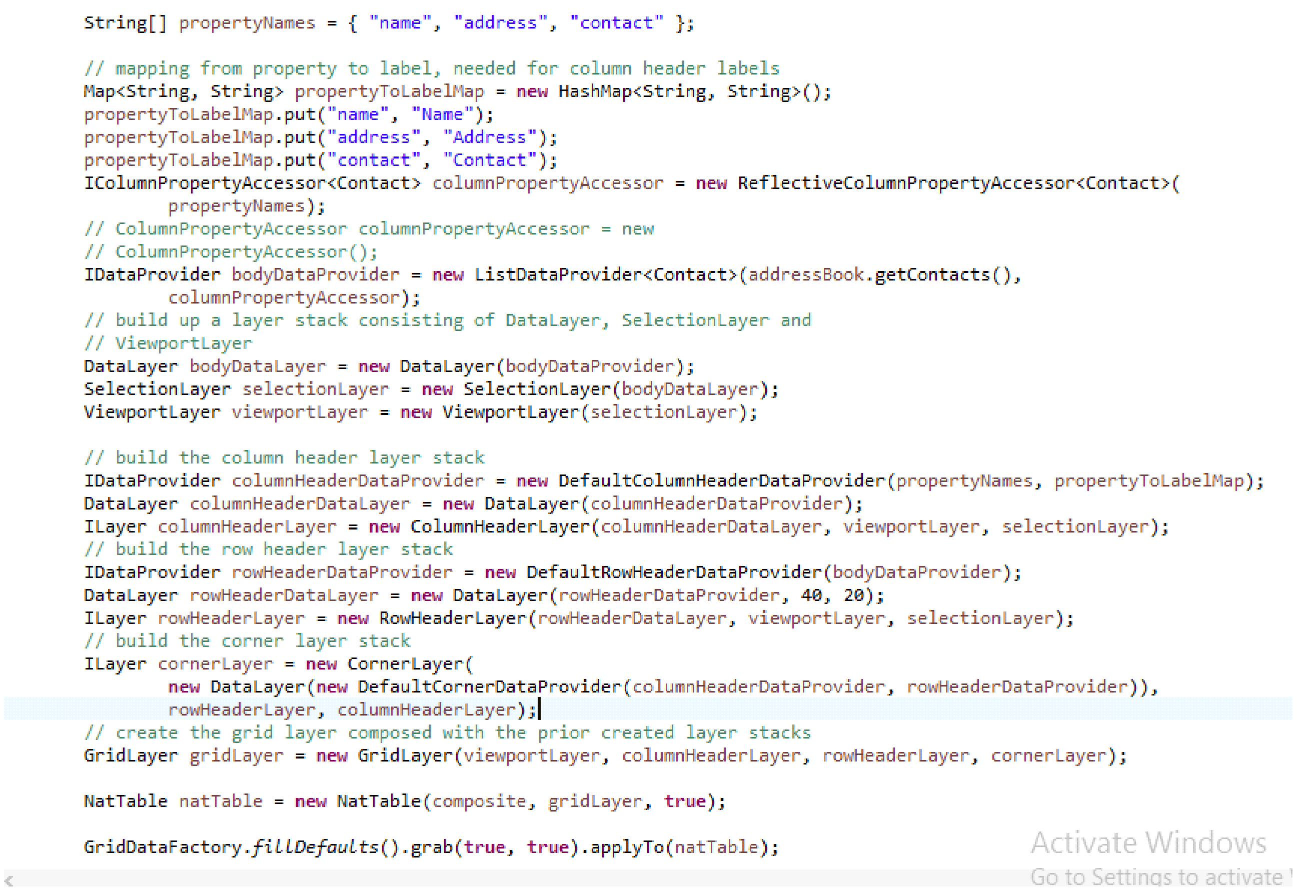
The row header is similar the column header. Note that the data layer also tracks the sizes of the rows/columns. Hence, you can set the default sizes in the constructor for the data layer.

Setting up the corner layer

The corner layer derives all its feature set from the column and row header layers. Hence, it can be set up very simply by passing in the dependents.

Drum roll ... Setting up the Grid Layer

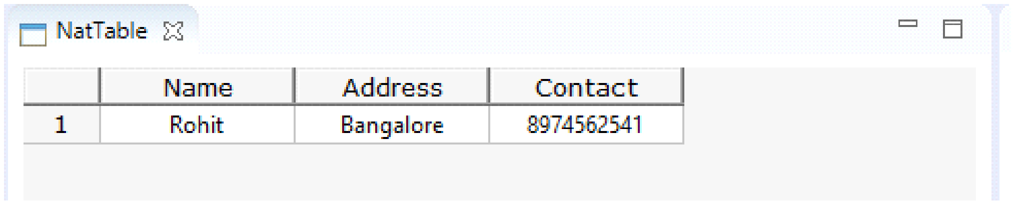
Now we have setup layer stacks for all regions in the grid. These stacks need to be unified to work as a coherent whole. We do this by placing a grid layer on the top. This layer is set as the underlying layer for NatTable and we are all ready to go.



```
String[] propertyNames = { "name", "address", "contact" };

// mapping from property to label, needed for column header labels
Map<String, String> propertyToLabelMap = new HashMap<String, String>();
propertyToLabelMap.put("name", "Name");
propertyToLabelMap.put("address", "Address");
propertyToLabelMap.put("contact", "Contact");
IColumnPropertyAccessor<Contact> columnPropertyAccessor = new ReflectiveColumnPropertyAccessor<Contact>(
        propertyNames);
// ColumnPropertyAccessor columnPropertyAccessor = new
// ColumnPropertyAccessor();
IDataProvider bodyDataProvider = new ListDataProvider<Contact>(addressBook.getContacts(),
        columnPropertyAccessor);
// build up a layer stack consisting of DataLayer, SelectionLayer and
// ViewportLayer
DataLayer bodyDataLayer = new DataLayer(bodyDataProvider);
SelectionLayer selectionLayer = new SelectionLayer(bodyDataLayer);
ViewportLayer viewportLayer = new ViewportLayer(selectionLayer);

// build the column header layer stack
IDataProvider columnHeaderDataProvider = new DefaultColumnHeaderDataProvider(propertyNames, propertyToLabelMap);
DataLayer columnHeaderDataLayer = new DataLayer(columnHeaderDataProvider);
ILayer columnHeaderLayer = new ColumnHeaderLayer(columnHeaderDataLayer, viewportLayer, selectionLayer);
// build the row header layer stack
IDataProvider rowHeaderDataProvider = new DefaultRowHeaderDataProvider(bodyDataProvider);
DataLayer rowHeaderDataLayer = new DataLayer(rowHeaderDataProvider, 40, 20);
ILayer rowHeaderLayer = new RowHeaderLayer(rowHeaderDataLayer, viewportLayer, selectionLayer);
// build the corner layer stack
ILayer cornerLayer = new CornerLayer(
        new DataLayer(new DefaultCornerDataProvider(columnHeaderDataProvider, rowHeaderDataProvider)),
        rowHeaderLayer, columnHeaderLayer);
// create the grid layer composed with the prior created layer stacks
GridLayer gridLayer = new GridLayer(viewportLayer, columnHeaderLayer, rowHeaderLayer, cornerLayer);

NatTable natTable = new NatTable(composite, gridLayer, true);

GridDataFactory.fillDefaults().grab(true, true).applyTo(natTable);
```

Activate Windows
Go to Settings to activate

Run the Application :



NatTable

	Name	Address	Contact
1	Rohit	Bangalore	8974562541

Changing AutoConfig to False in NatTable Constructor & Adding default style and header menus using autoconfiguration meathod.

Here you will find general information about configuring NatTable and the concepts involved.

ConfigRegistry

This is a global object holding the following kinds of configuration

- Styling
- Editing
- Comparators for sorting
- Any piece of arbitary information can be stored in this registry.

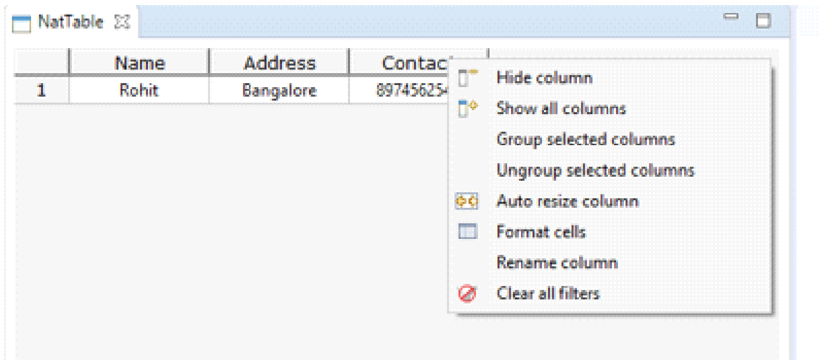
UiBindingRegistry

This is a global object holding the following kinds of configuration

- Key bindings
- Mouse bindings

```
NatTable natTable = new NatTable(composite, gridLayer, false);
GridDataFactory.fillDefaults().grab(true, true).applyTo(natTable);
natTable.addConfiguration(new DefaultNatTableStyleConfiguration());
natTable.addConfiguration(new HeaderMenuConfiguration(natTable));
natTable.configure();
```

Run the Application :



NatTable

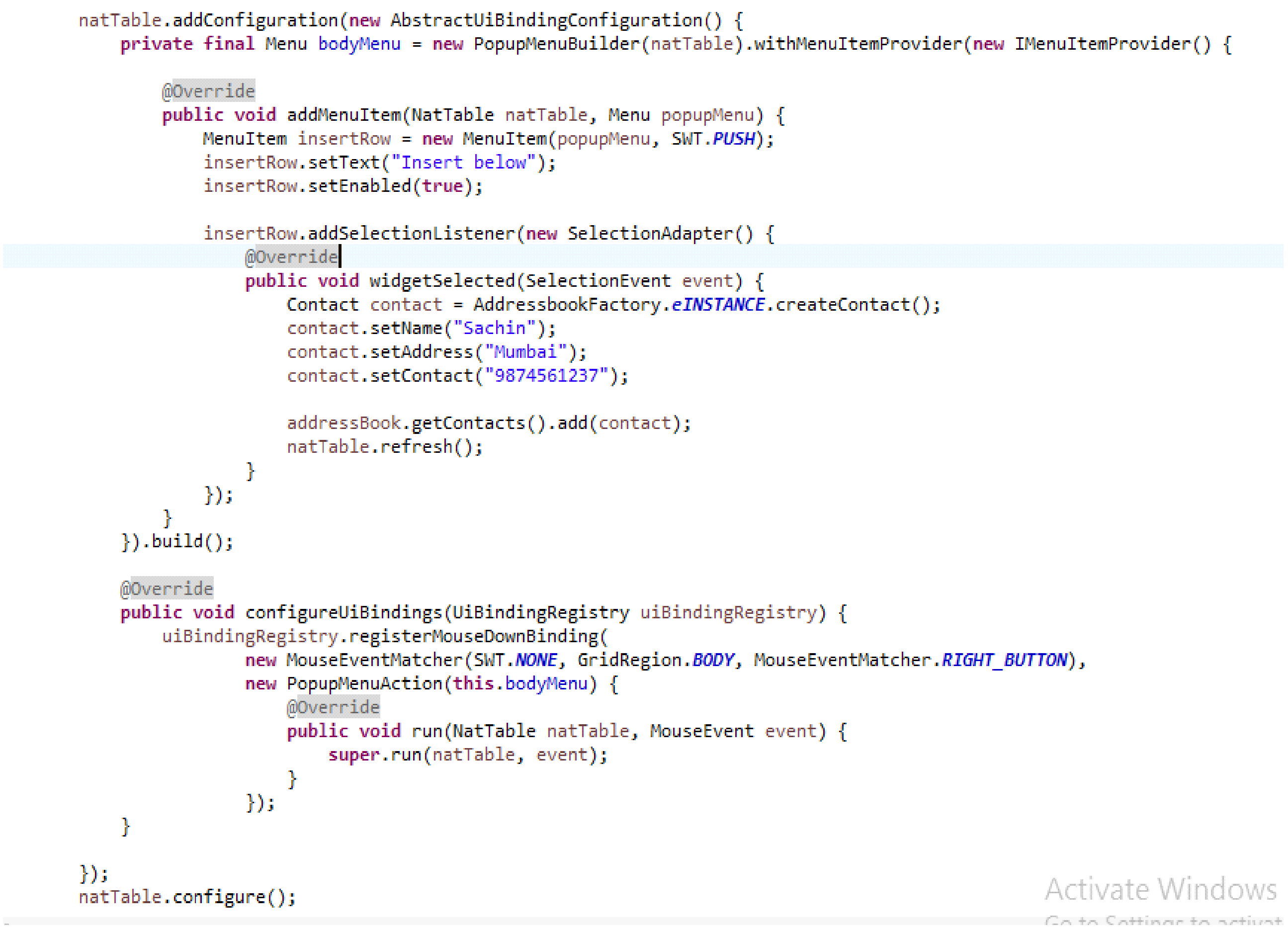
	Name	Address	Contac
1	Rohit	Bangalore	897456254

Adding Popup Menu to NatTable:

The `PopupMenuBuilder` is a builder in NatTable to create a menu with menu items that perform NatTable commands. It has several methods for adding such menu items and initializes and returns the menu on calling `PopupMenuBuilder#build()`.
To create a menu with NatTable commands you need to perform the following steps:

- Create an `IConfiguration` for the menu by extending `AbstractUiBindingConfiguration`
- Create a menu using the NatTable `PopupMenuBuilder` helper class
- Register a `PopupMenuAction` binding using the created menu
- Add the `IConfiguration` to the NatTable instance

The following code shows the `DebugMenuConfiguration` that is shipped with NatTable to add debugging capability in a rendered NatTable.

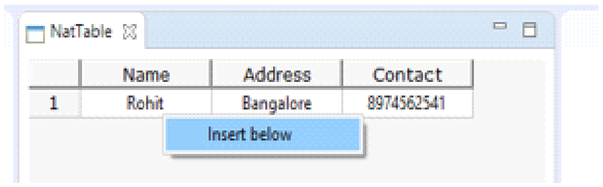


```
natTable.addConfiguration(new AbstractUiBindingConfiguration() {
    private final Menu bodyMenu = new PopupMenuBuilder(natTable).withMenuItemProvider(new IMenuItemProvider() {

        @Override
        public void addMenuItem(NatTable natTable, Menu popupMenu) {
            MenuItem insertRow = new MenuItem(popupMenu, SWT.PUSH);
            insertRow.setText("Insert below");
            insertRow.setEnabled(true);

            insertRow.addSelectionListener(new SelectionAdapter() {
                @Override
                public void widgetSelected(SelectionEvent event) {
                    Contact contact = AddressbookFactory.eINSTANCE.createContact();
                    contact.setName("Sachin");
                    contact.setAddress("Mumbai");
                    contact.setContact("9874561237");

                    addressBook.getContacts().add(contact);
                    natTable.refresh();
                }
            });
        }
    }).build();

    @Override
    public void configureUiBindings(UiBindingRegistry uiBindingRegistry) {
        uiBindingRegistry.registerMouseDownBinding(
                new MouseEventMatcher(SWT.NONE, GridRegion.BODY, MouseEventMatcher.RIGHT_BUTTON),
                new PopupMenuAction(this.bodyMenu) {
                    @Override
                    public void run(NatTable natTable, MouseEvent event) {
                        super.run(natTable, event);
                    }
                });
    }

});
natTable.configure();
```

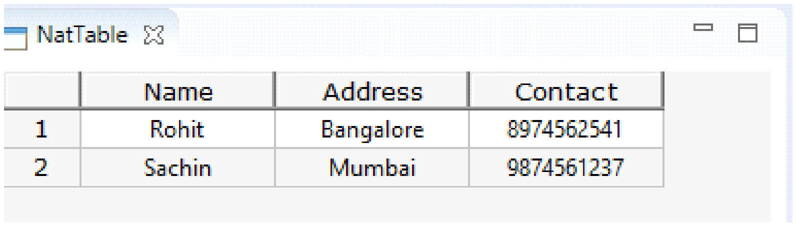
Activate Windows
Go to Settings to activat

Run the Application :

ANCIT CONSULTING



	Name	Address	Contact
1	Rohit	Bangalore	8974562541



	Name	Address	Contact
1	Rohit	Bangalore	8974562541
2	Sachin	Mumbai	9874561237

Adding Row Selection Provider:

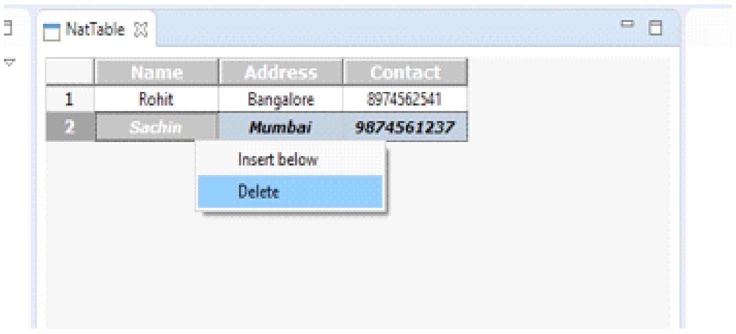
Implementation of ISelectionProvider to add support for JFace selection handler.
The SelectionLayer this ISelectionProvider is connected to.
The IRowDataProvider to access the selected row data.

```
final RowSelectionProvider<Contact> selectionProvider = new RowSelectionProvider<Contact>(selectionLayer, bodyDataProvider);
```

Context menu to delete selected row

```
}).withMenuItemProvider(new IMenuItemProvider() {

    @Override
    public void addMenuItem(NatTable natTable, Menu popupMenu) {
        MenuItem deleteRow = new MenuItem(popupMenu, SWT.PUSH);
        deleteRow.setText("Delete");
        deleteRow.setEnabled(true);

        deleteRow.addSelectionListener(new SelectionAdapter() {
            @Override
            public void widgetSelected(SelectionEvent event) {
                IStructuredSelection selection = (IStructuredSelection) selectionProvider.getSelection();
                @SuppressWarnings("unchecked")
                List<Contact> contacts = selection.toList();
                addressBook.getContacts().removeAll(contacts);
                natTable.refresh();

            }
        });
    }
}).build();
```

Run the Application :



	Name	Address	Contact
1	Rohit	Bangalore	8974562541
2	Sachin	Mumbai	9874561237

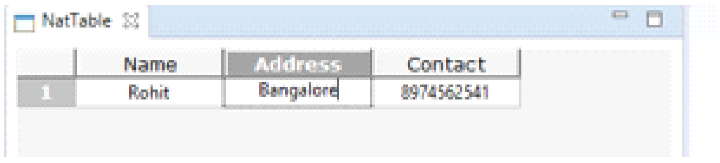
Adding Editor Configuration to edit cell in the NatTable:

Now for simple text editing purposes only
a EditConfigAttributes.CELL_EDITABLE_RULE config attribute has to be registered to the IConfigRegistry.

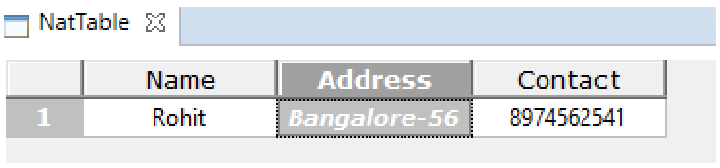
```
natTable.addConfiguration(new NatTableEditConfig());

natTable.configure();
```

```
public class NatTableEditConfig extends AbstractRegistryConfiguration {

    @Override
    public void configureRegistry(IConfigRegistry configRegistry) {
        configRegistry.registerConfigAttribute(EditConfigAttributes.CELL_EDITABLE_RULE, IEditableRule.ALWAYS_EDITABLE);
    }

}
```

Run the Application :



	Name	Address	Contact
1	Rohit	Bangalore	8974562541



	Name	Address	Contact
1	Rohit	Bangalore-56	8974562541

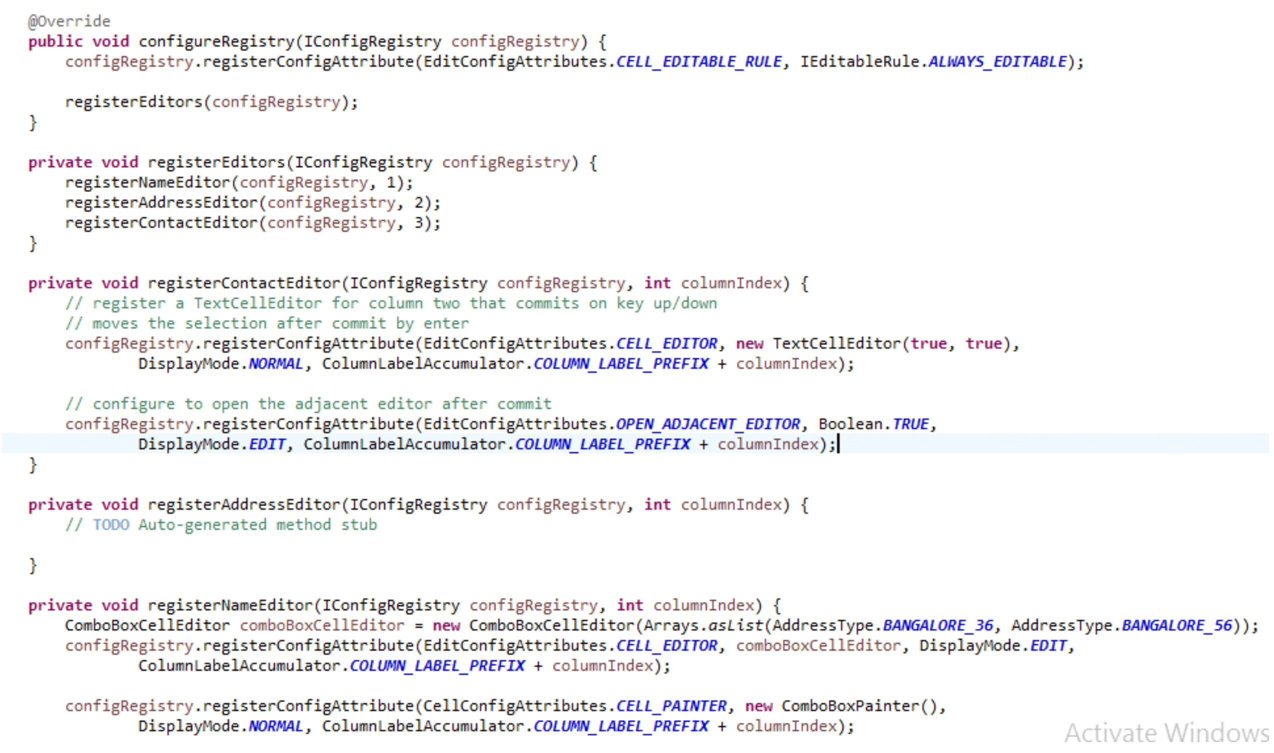
Add more advanced editing support to a HYPERLINK "http://www.vogella.com/tutorials/NatTableEditing/article.html" HYPERLINK "http://www.vogella.com/tutorials/NatTableEditing/article.html" HYPERLINK "http://www.vogella.com/tutorials/NatTableEditing/article.html"NatTable.

ANCIT CONSULTING

In this exercise editing support will be provided for a table, which contains different data than only strings. So different cell editors have to be applied for different columns.

For the `SimpleEditor` an anonymous inner class of an `AbstractRegistryConfiguration` was used to apply basic editing capabilities.

Now that the configuration becomes more complex it is better to encapsulate it in another class:



```
@Override
public void configureRegistry(IConfigRegistry configRegistry) {
    configRegistry.registerConfigAttribute(EditConfigAttributes.CELL_EDITABLE_RULE, IEditableRule.ALWAYS_EDITABLE);

    registerEditors(configRegistry);
}

private void registerEditors(IConfigRegistry configRegistry) {
    registerNameEditor(configRegistry, 1);
    registerAddressEditor(configRegistry, 2);
    registerContactEditor(configRegistry, 3);
}

private void registerContactEditor(IConfigRegistry configRegistry, int columnIndex) {
    // register a TextCellEditor for column two that commits on key up/down
    // moves the selection after commit by enter
    configRegistry.registerConfigAttribute(EditConfigAttributes.CELL_EDITOR, new TextCellEditor(true, true),
            DisplayMode.NORMAL, ColumnLabelAccumulator.COLUMN_LABEL_PREFIX + columnIndex);

    // configure to open the adjacent editor after commit
    configRegistry.registerConfigAttribute(EditConfigAttributes.OPEN_ADJACENT_EDITOR, Boolean.TRUE,
            DisplayMode.EDIT, ColumnLabelAccumulator.COLUMN_LABEL_PREFIX + columnIndex);
}

private void registerAddressEditor(IConfigRegistry configRegistry, int columnIndex) {
    // TODO Auto-generated method stub

}

private void registerNameEditor(IConfigRegistry configRegistry, int columnIndex) {
    ComboBoxCellEditor comboBoxCellEditor = new ComboBoxCellEditor(Arrays.asList(AddressType.BANGALORE_36, AddressType.BANGALORE_56));
    configRegistry.registerConfigAttribute(EditConfigAttributes.CELL_EDITOR, comboBoxCellEditor, DisplayMode.EDIT,
            ColumnLabelAccumulator.COLUMN_LABEL_PREFIX + columnIndex);

    configRegistry.registerConfigAttribute(CellConfigAttributes.CELL_PAINTER, new ComboBoxPainter(),
            DisplayMode.NORMAL, ColumnLabelAccumulator.COLUMN_LABEL_PREFIX + columnIndex);
```

Activate Windows

ABOUT ANCIT:

ANCIT Consulting is an Eclipse Consulting Firm located in the "Silicon Valley of Outsourcing", Bangalore. Offers professional Eclipse Support and Training for various Eclipse based Frameworks including RCP, EMF, GEF, GMF. Contact us on annamalai@ancitconsulting.com to learn more about our services.

ANCIT CONSULTING