

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 natable.

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

```

- @Override
  public int getColumnCount() {

      return 3;
  }

- @Override
  public String getColumnProperty(int columnIndex) {

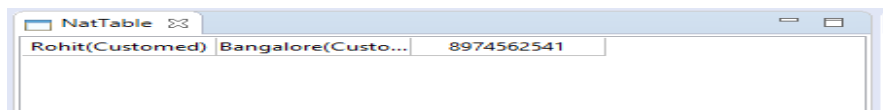
      return propertyNames.get(columnIndex);
  }

- @Override
  public int getColumnIndex(String propertyName) {

      return propertyNames.indexOf(propertyName);
  }
}

```

Run the Application :



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);

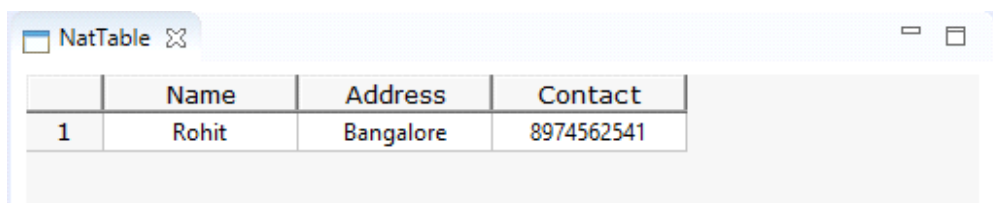
NatTable natTable = new NatTable(composite, gridLayer, true);

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

```

Activate Windows
Go to Settings to activate!

Run the Application :



	Name	Address	Contact
1	Rohit	Bangalore	8974562541

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.