JFace Data Binding - Tutorial

Lars Vogel

Version 3.0

02.08.2013

JFace Data binding

This tutorial explains Eclipse JFace Data Binding which can be used to synchronize data between different objects. This tutorial is based on Eclipse 4.3 and Java 1.7.

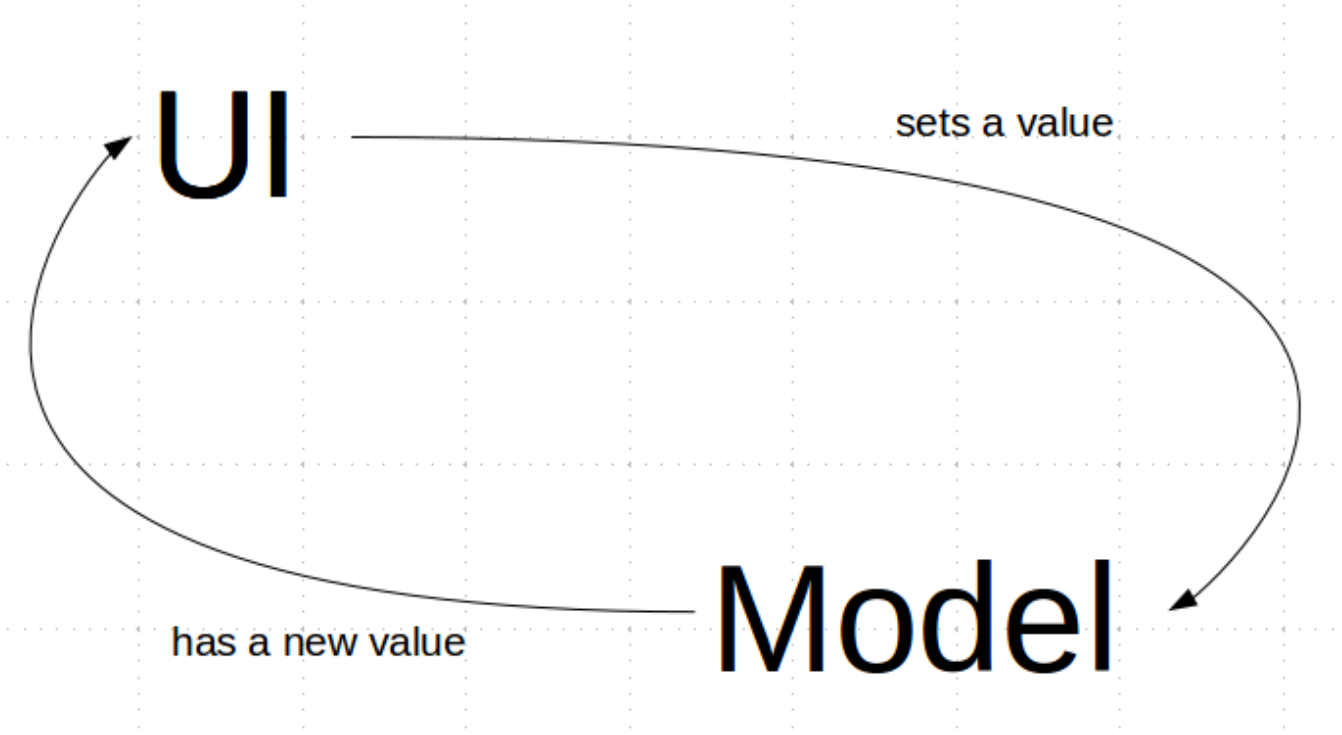
Table of Contents

1. Data Binding with JFace

1.1. What are Data Binding frameworks?

A *Data Binding* framework connects properties of objects. It is typically used to synchronize properties of user interface widgets with properties of other Java objects. These Java objects are typically called the *data model* or the *domain model*.

Data Binding frameworks synchronize changes of these properties. They typically support to include validation and conversion into the synchronization process. This synchronization is depicted in the following graphic.



1.2. JFace Data Binding

JFace Data Binding is an Eclipse framework for data binding. JFace Data Binding is mainly used in Eclipse plug-ins. It has also been ported to be used in other frameworks for example the *Google Web Toolkit* (GWT) user interface toolkit.

For example you could bind the String property called *firstName* of a Java object to a *text* property of the SWT `Text` widget. If the user changes the text in the user interface, the corresponding property in the Java object is updated.

1.3. Reacting to changes via property change listeners

To be able to react to changes in an attribute of a Java objects, JFace Data Binding needs to be able to register itself as a listener to the attribute. The SWT and JFace widgets support this.

JFace Data Binding can be used to observe attributes of a domain model. The JFace Data Binding framework can register listeners for these Java objects and gets notified if a change in the model happens. This change notification from the domain model requires that the model objects implement property change support.

2. Creating bindings

2.1. Java POJO vs Java Bean

The data model is typically described as a *Java Plain Old Java Object (POJO)* model or a *Java Bean* model.

The term POJO is used to describe a Java object which does not follow any of the major Java object models, conventions, or frameworks, i.e. a Java object which does not have to fulfill any specific requirements. For example the following is a POJO.

```java
package de.vogella.databinding.example;

public class Person  {
  private String name;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

A Java Bean is a Java object which follow the Java Bean specification. This specification requires that the class implements `getter` and `setter` methods for all its attributes. It must also implement property change support via the `PropertyChangeSupport` class and propagate changes to registered listeners.

A Java class which provides `PropertyChangeSupport` looks like the following example.

```java
package de.vogella.databinding.example;

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

public class ModelObject {
  private PropertyChangeSupport changeSupport =
      new PropertyChangeSupport(this);

  public void addPropertyChangeListener(PropertyChangeListener
      listener) {
    changeSupport.addPropertyChangeListener(listener);
  }

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

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

  public void removePropertyChangeListener(String propertyName,
      PropertyChangeListener listener) {
    changeSupport.removePropertyChangeListener(propertyName, listener);
  }

  protected void firePropertyChange(String propertyName,
      Object oldValue,
      Object newValue) {
    changeSupport.firePropertyChange(propertyName, oldValue, newValue);
  }
}
```

Other domain classes could extend this class. The following example demonstrates that.

```java
package de.vogella.databinding.example;

public class Person extends ModelObject {
  private String name;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    firePropertyChange("name", this.name, this.name = name);
  }
}
```

2.2. Data Binding and POJOs

Data Binding supports a POJO domain model. A POJO does not support change notification. In this case changes in the domain model are not propagated to the user interface.

You can still use Data Binding to connect the user interface to the data model. In this case changes in the user interface are propagated to the domain model.

2.3. Observable

JFace Data Binding allows you to observe arbitrary attributes. For widgets you typically observe the *text* property but you can also observe other values. For example, you could bind the *enabled* property to a boolean value of the data model.

The `IObservableValue` interface is used to observe properties of objects.

JFace Data Binding contains the *Properties API* as the recommended way of using the framework. The *Properties API* provides factories to create `IObservableValue` objects.

The main factories are `PojoProperties`, `BeanProperties`, `WidgetProperties` and `ViewerProperties`.

Table 1. Factories

Factory	Description
`PojoProperties`	Used to create `IObservableValues` for Java objects.
`BeanProperties`	Used to create `IObservableValues` objects for Java Beans.
`WidgetProperties`	Used to create `IObservableValues` for properties of SWT widgets.
`ViewerProperties`	Used to create `IObservableValues` for properties of JFace Viewer.
`Properties`	Used to create `IObservables` for properties of any type like Objects, Collections or Maps.
`Observables`	Used to create `IObservables` for properties of special Objects, Collections, Maps and Entries of an IObservableMap.

2.4. How to observe properties

The following code demonstrates how to create an `IObservableValue` object for the *firstName* property of a Java object called *person*.

```
// if person is a POJO
IObservableValue myModel = PojoProperties.value("firstName").
  observe(person)

// alternatively if person is a bean use
// prefer using beans if you data model provides property change support
IObservableValue myModel = BeansProperties.value("firstName").
  observe(person)
```

The next example demonstrates how to create an `IObservableValue` for the *text* property of an SWT `Text` widget called *firstNameText*.

```
IObservableValue target = WidgetProperties.text(SWT.Modify).
  observe(firstNameText);
```

In case you do not want to manipluate the object's properties directly or you got a more generic datamodel, where you do not know which attributes should be bound beforehand, you could place those attributes into an `IObservableMap` and observe the MapEntries with the Observables.observeMapEntry() method.

```
import javax.annotation.PostConstruct;
import javax.inject.Inject;
import javax.inject.Named;

import org.eclipse.core.databinding.DataBindingContext;
import org.eclipse.core.databinding.observable.Observables;
import org.eclipse.core.databinding.observable.map.IObservableMap;
import org.eclipse.core.databinding.observable.map.WritableMap;
import org.eclipse.core.databinding.observable.value.IObservableValue;
import org.eclipse.e4.core.di.annotations.Optional;
import org.eclipse.e4.ui.di.Persist;
import org.eclipse.e4.ui.services.IServiceConstants;
import org.eclipse.jface.databinding.swt.ISWTObservableValue;
import org.eclipse.jface.databinding.swt.WidgetProperties;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Text;

import com.example.e4.rcp.todo.model.Todo;

public class ObservableMapEntry {

  private static final String SECOND_ATTRIBUTE = "secondAttribute";
  private static final String FIRST_ATTRIBUTE = "firstAttribute";

  private IObservableMap attributesMap = new WritableMap();
  private DataBindingContext dbc;
  private Todo todo;

  @PostConstruct
  public void createUI(Composite parent) {
    dbc = new DataBindingContext();

    Text txtFirstAttribute = new Text(parent, SWT.BORDER);
    Text txtSecondAttribute = new Text(parent, SWT.BORDER);

    // create observables for the Text controls
    ISWTObservableValue txtFirstAttributeObservable = WidgetProperties.text(SWT.Modify).observe(txtFirstAttribute);
    ISWTObservableValue txtSecondAttributeObservable = WidgetProperties.text(SWT.Modify)
        .observe(txtSecondAttribute);

    // create observables for the Map entries
    IObservableValue firstAttributeObservable = Observables.observeMapEntry(attributesMap, FIRST_ATTRIBUTE);
    IObservableValue secondAttributeObservable = Observables.observeMapEntry(attributesMap, SECOND_ATTRIBUTE)
;

    dbc.bindValue(txtFirstAttributeObservable, firstAttributeObservable);
```

```
    dbc.bindValue(txtSecondAttributeObservable, secondAttributeObservable);
  }

  @Inject
  @Optional
  public void setModel(@Named(IServiceConstants.ACTIVE_SELECTION) Todo todo) {
    if (todo != null) {
      this.todo = todo;
      // Set new values for the map entries from a model object
      attributesMap.put(FIRST_ATTRIBUTE, todo.getSummary());
      attributesMap.put(SECOND_ATTRIBUTE, todo.getDescription());
    }
  }

  @Persist
  public void save() {
    if (todo != null) {
      // only store the actual values on save and not directly
      todo.setSummary((String) attributesMap.get(FIRST_ATTRIBUTE));
      todo.setDescription((String) attributesMap.get(SECOND_ATTRIBUTE));
    }
  }
}
```

2.5. Observing nested properties

You can also observe nested model properties, e.g. attributes of classes which are contained in another class.

The following code demonstrates how to access the *country* property in the *address* field of the object *person*.

```
IObservable model = PojoProperties.value(Person.class,
    "address.country").observe(person);
```

2.6. DataBindingContext

The `DataBindingContext` class provides the functionality to connect `IObservableValues` objects.

Via the `DataBindingContext.bindValue()` method two `IObservableValues` objects are connected. The first parameter is the target and the second is the model. During the initial binding the value from the model is copied to the target.

```
// create new Context
DataBindingContext ctx = new DataBindingContext();

// define the IObservables
IObservableValue target = WidgetProperties.text(SWT.Modify).
  observe(firstName);
IObservableValue model= BeanProperties.
  value(Person.class,"firstName").observe(person);

// connect them
ctx.bindValue(target, model);
```

> **Note:** The initial copying from model to target is useful for the initial synchronization. For example if you have an attribute of a `Person` p object and the text attribute of a `Text` txtName widget, you typically want to copy the value from p to txtName at the beginning.

3. JFace Data Binding Plug-ins

In the *MANIFEST.MF*, add the following plug-ins as dependencies to your plug-in to use JFace Data Binding.

- org.eclipse.core.databinding
- org.eclipse.core.databinding.beans
- org.eclipse.core.databinding.property
- org.eclipse.jface.databinding

4. More on bindings

4.1. UpdateValueStrategy

The `bindValue()` method allows you to specify `UpdateValueStrategy` objects as third and fourth parameters. These objects allow you to control how and when the values are updated. If no `UpdateValueStrategy` is specified, defaults will be used.

4.2. Converter and Validator

Validators allow you to implement validation of the data before it is propagated to the other connected property. A class which wants to provide this functionality must implement the `org.eclipse.core.databinding.validation.IValidator` interface.

Converters allow you to convert the values between the model and the target. Converters are defined based on the `IConverter` interface.

Instances of these objects can be added to the `UpdateValueStrategy` object.

```
// define a validator to check that only numbers are entered
IValidator validator = new IValidator() {
  @Override
  public IStatus validate(Object value) {
    if (value instanceof Integer) {
      if (value.toString().matches(".*\\d.*")) {
        return ValidationStatus.ok();
      }
    }
    return ValidationStatus.error(value.toString() +"is not a number");
  }
};

// create UpdateValueStrategy and assign
// to the binding
UpdateValueStrategy strategy = new UpdateValueStrategy();
strategy.setBeforeSetValidator(validator);

Binding bindValue =
  ctx.bindValue(widgetValue, modelValue, strategy, null);
```

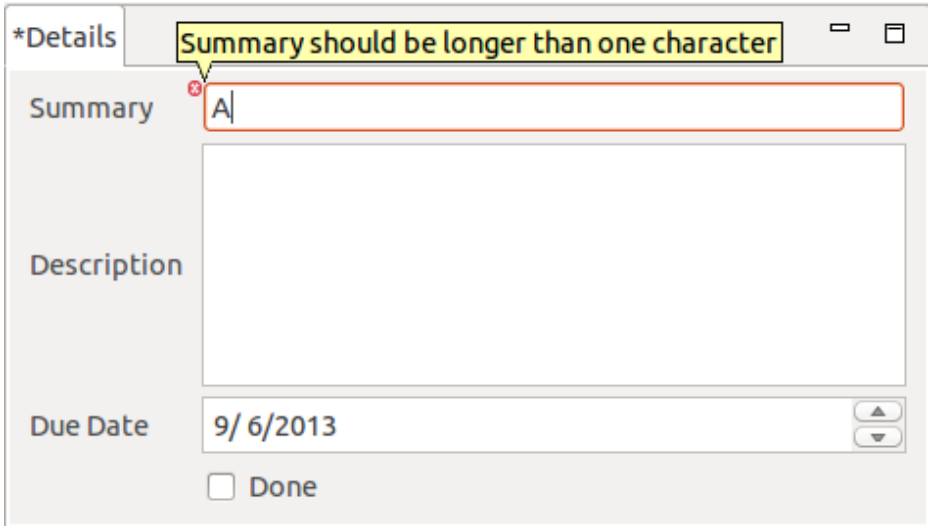
> **Tip:** The `WizardPageSupport` class provides support to connect the result from the given data binding context to the given wizard page, updating the wizard page's completion state and its error message accordingly.

4.3. ControlDecorators

JFace Data Binding allows you to use icon decorators in the user interface which reflect the status of the field validation. This allows you to provide immediate feedback to the user. For the creation of the control decoration you use the return object from the `bindvalue()` method of `DataBindingContext` object.

```
// The following code assumes that a Validator is already defined
Binding bindValue =
  ctx.bindValue(widgetValue, modelValue, strategy, null);

// add some decorations to the control
ControlDecorationSupport.create(bindValue, SWT.TOP | SWT.LEFT);
```

The result might look like the following screenshot.



4.4. Placeholder binding with WritableValue

You can create bindings to a `WritableValue` object. A `WritableValue` object can hold a reference to another object.

You can exchange this reference in `WritableValue` and the databinding will use the new (reference) object for its binding. This way you can create the binding once and still exchange the object which is bound by databinding.

To bind to a `WritableValue` you use the `observeDetail()` method, to inform the framework that you would like to observe the contained object.

```
WritableValue value  = new WritableValue();

// create the binding
DataBindingContext ctx = new DataBindingContext();
IObservableValue target = WidgetProperties.
  text(SWT.Modify).observe(text);
IObservableValue model = BeanProperties.value("firstName").
    observeDetail(value);

ctx.bindValue(target, model);

// create a Person object called p
Person p = new Person();

// make the binding valid for this new object
value.setValue(p);
```

You may also see that by using observeDetail() you can already create a binding even in case the actual value to be observed is not present at the moment.

4.5. Listening to all changes in the binding

You can register a listener to all bindings of the `DataBindingContext` class. Your listener will be called when something has changed.

For example this can be used to determine the status of a part which behaves like an editor. If something is changed, it will mark itself as dirty.

```
// define your change listener
// dirty holds the state for the changed status of the editor
IChangeListener listener = new IChangeListener() {
  @Override
  public void handleChange(ChangeEvent event) {
    // Ensure dirty is not null
    if (dirty!=null){
      dirty.setDirty(true);
    }
};


private void updateUserInterface(Todo todo) {

  // check that user interface is available
  if (summary != null && !summary.isDisposed()) {


    // Deregister change listener to the old binding
    IObservableList providers = ctx.getValidationStatusProviders();
    for (Object o : providers) {
      Binding b = (Binding) o;
      b.getTarget().removeChangeListener(listener);
    }

    // dispose the binding
    ctx.dispose();

    // NOTE
    // HERE WOULD BE THE CODE DATABINDING CODE
    // INTENTIALLY LEFT OUT FOR BREVITY



    // get the validation status provides
    IObservableList bindings =
      ctx.getValidationStatusProviders();

    // mot all validation status providers
    // are bindings, e.g. MultiValidator
    // otherwise you could use
    // context.getBindings()

    // register the listener to all bindings
    for (Object o : bindings) {
      Binding b = (Binding) o;
      b.getTarget().addChangeListener(listener);
    }
  }
}
```

4.6. More information on Data Binding

Data Binding provides lots of examples for other use cases via its version control system. See **Wiki on JFace**

Data Binding for instructions how to access this information.

5. Data Binding for JFace Viewers

5.1. Binding Viewers

JFace Data Binding provides functionality to bind the data of JFace viewers , e.g. for `TableViewers`.

Data binding for these viewers distinguish between changes in the collection and changes in the individual object.

In the case that Data Binding observes a collection, it requires a `ContentProvider` which notifies it, once the data in the collection changes.

`ObservableListContentProvider` is a `ContentProvider` which requires a list implementing the `IObservableList` interface. The `Properties` class allows you to wrap another list with its `selfList()` method into an `IObservableList`.

The following snippet demonstrates the usage:

```
// use ObservableListContentProvider
viewer.setContentProvider(new ObservableListContentProvider());

// create sample data
List<Person> persons = createExampleData();

// wrap the input into a writable list
IObservableList input =
  Properties.selfList(Person.class).observe(persons);

// set the IObservableList as input for the viewer
viewer.setInput(input);
```

5.2. Observing list details

You can also use the `ObservableMapLabelProvider` class to observe changes of the list elements.

```
ObservableListContentProvider contentProvider =
  new ObservableListContentProvider();

// create the label provider including monitoring
// of the changes of the labels
IObservableSet knownElements = contentProvider.getKnownElements();

final IObservableMap firstNames = BeanProperties.value(Person.class,
  "firstName").observeDetail(knownElements);
final IObservableMap lastNames = BeanProperties.value(Person.class,
  "lastName").observeDetail(knownElements);

IObservableMap[] labelMaps = { firstNames, lastNames };

ILabelProvider labelProvider =
  new ObservableMapLabelProvider(labelMaps) {
  public String getText(Object element) {
    return firstNames.get(element) + " " + lastNames.get(element);
    }
  };
```

5.3. ViewerSupport

`ViewerSupport` simplifies the setup for JFace viewers in cases where all columns should be displayed. It registers changes listener on the collection as well as on the individual elements.

`ViewerSupport` creates via the `bind()` method the `LabelProvider` and `ContentProvider` for a viewer automatically.

```
// the MyModel.getPersons() method call returns a List<Person> object
// the WritableList object wraps this object in an IObservableList

input = new WritableList(MyModel.getPersons(), Person.class);

// The following  creates and binds the data
// for the Table based on the provided input
// no additional label provider /
// content provider / setInput required

ViewerSupport.bind(viewer, input,
    BeanProperties.
    values(new String[] { "firstName", "lastName", "married" }));
```

5.4. ObservableValueEditingSupport

Also when editing cells in a `Table` or `Tree` databinding is your friend. You can apply an `ObservableValueEditingSupport` to your `TableViewerColumn` or `TreeViewerColumn`.

In the following example we create such an `ObservableValueEditingSupport` where an observable of the `TextCellEditor` control and an observable for the *firstname* property is created.

```java
import org.eclipse.core.databinding.DataBindingContext;
import org.eclipse.core.databinding.beans.BeanProperties;
import org.eclipse.core.databinding.observable.value.IObservableValue;
import org.eclipse.jface.databinding.swt.WidgetProperties;
import org.eclipse.jface.databinding.viewers.ObservableValueEditingSupport;
import org.eclipse.jface.viewers.CellEditor;
import org.eclipse.jface.viewers.ColumnViewer;
import org.eclipse.jface.viewers.TextCellEditor;
import org.eclipse.jface.viewers.ViewerCell;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;

public class MyObservableEditingSupport extends ObservableValueEditingSupport {

  public MyObservableEditingSupport(ColumnViewer viewer,
      DataBindingContext dbc) {
    super(viewer, dbc);
  }

  @Override
  protected IObservableValue doCreateCellEditorObservable(CellEditor cellEditor) {
    return WidgetProperties.text(SWT.Modify).observe(cellEditor.getControl());
  }

  @Override
  protected IObservableValue doCreateElementObservable(Object element,
      ViewerCell cell) {
    return BeanProperties.value("firstName").observe(element);
  }

  @Override
  protected CellEditor getCellEditor(Object element) {
    return new TextCellEditor((Composite) getViewer().getControl());
  }

}
```

`MyObservableEditingSupport` can now be applied to a `TableViewerColumn`.

```java
DataBindingContext dbc = new DataBindingContext();

TableViewer personViewer = new TableViewer(parent);
personViewer.getTable().setHeaderVisible(true);

TableViewerColumn firstNameViewerColumn = new TableViewerColumn(personViewer, SWT.NONE);
firstNameViewerColumn.getColumn().setText("First name");
firstNameViewerColumn.getColumn().setWidth(300);

// apply MyObservableEditingSupport to the first name TableViewerColumn
firstNameViewerColumn.setEditingSupport(new MyObservableEditingSupport(personViewer, dbc));

ViewerSupport.bind(viewer, input,
    BeanProperties.value(String.class, "firstName"));
```

5.5. Master Detail binding

The `ViewerProperties` class allows you to create `IObservableValues` for properties of the viewer. For example you can track the current selection, e.g. which data object is currently selected. This binding is called *Master Detail* binding as you track the selection of a master.

To access fields in the selection you can use the `PojoProperties` or the `BeanProperties` class. Both provide the `value().observeDetail()` method chain, which allows you to observe a detail value of an `IObservableValues` object.

For example the following will map the *summary* property of the `Todo` domain object to a `Label` based on the selection of a `ComboViewer`.

```
// assume we have Todo domain objects
// todos is a of type: List<Todo>
final ComboViewer viewer = new ComboViewer(parent, SWT.DROP_DOWN);
viewer.setContentProvider(ArrayContentProvider.getInstance());
viewer.setLabelProvider(new LabelProvider() {
  public String getText(Object element) {
    Todo todo = (Todo) element;
    return todo.getSummary();
  };
});
viewer.setInput(todos);

// create a Label to map to
Label label = new Label(parent, SWT.BORDER);
// parent has a GridLayout assigned
label.setLayoutData(new GridData(SWT.FILL, SWT.CENTER, true, false));


DataBindingContext dbc = new DataBindingContext();

// for binding to the label
IObservableValue target = WidgetProperties.text().observe(label);

// observe the selection
IViewerObservableValue selectedTodo = ViewerProperties
    .singleSelection().observe(viewer);
// observe the summary attribute of the selection
IObservableValue detailValue =
  PojoProperties
    .value("summary", String.class)
    .observeDetail(selectedTodo)

dbc.bindValue(target, detailValue);
```

5.6. Chaining properties

You can chain properties together to simplify observing nested properties. The following code examples demonstrates this.

```
IObservableValue viewerSelectionSummaryObservable =
  ViewerProperties.singleSelection()
    .value(BeanProperties.value("summary", String.class))
    .observe(viewer);
```

```
IListProperty siblingNames = BeanProperties.
  value("parent").list("children").values("name");
IObservableList siblingNamesObservable =
  siblingNames.observe(node);
```

6. Prerequisites for this tutorial

This article assumes what you have basic understanding of development for the Eclipse platform. Please see **Eclipse RCP Tutorial** or **Eclipse Plugin Tutorial**.

For the databinding with JFace *Viewers* you should already be familiar with the concept of JFace *Viewers*.

For an introduction on JFace Viewers please see **JFace Overview**, **JFace Tables** and **JFace Trees**

7. Data Binding with SWT controls

7.1. First example

Create a new Eclipse RCP project "de.vogella.databinding.example" using the template "RCP application with a View".

Create the `de.vogella.databinding.person.model` package and the following model classes.

```
package de.vogella.databinding.example.model;

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

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

  public Person() {
  }

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

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

  public Address getAddress() {
    return address;
  }

  public void setAddress(Address address) {
    address.addPropertyChangeListener("country", this);
    propertyChangeSupport.firePropertyChange("address", this.address,
        this.address = address);
  }
```

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

  @Override
  public void propertyChange(PropertyChangeEvent event) {
    propertyChangeSupport.firePropertyChange("address", null, address);
  }

}
```

```
package de.vogella.databinding.example.model;

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

public class Address {

  private String street;
  private String number;
  private String postalCode;
  private String city;
  private String country;
  private PropertyChangeSupport propertyChangeSupport = new PropertyChangeSupport(this);

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

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

  public Address() {
  }

  public Address(String postalCode, String city, String country) {
    this.postalCode = postalCode;
    this.city = city;
    this.country = country;
  }

  public String getStreet() {
    return street;
  }

  public void setStreet(String street) {
    propertyChangeSupport.firePropertyChange("street", this.street,
        this.street = street);
  }

  public String getNumber() {
```

```
    return number;
  }

  public void setNumber(String number) {
    propertyChangeSupport.firePropertyChange("number", this.number,
        this.number = number);
  }

  public String getPostalCode() {
    return postalCode;
  }

  public void setPostalCode(String postalCode) {
    propertyChangeSupport.firePropertyChange("postalCode", this.postalCode,
        this.postalCode = postalCode);
  }

  public String getCity() {
    return city;
  }

  public void setCity(String city) {
    propertyChangeSupport.firePropertyChange("citry", this.city,
        this.city = city);
  }

  public String getCountry() {
    return country;
  }

  public void setCountry(String country) {
    propertyChangeSupport.firePropertyChange("country", this.country,
        this.country = country);
  }

  public String toString() {
    String s = "";
    s += street != null ? street + " " : "";
    s += number != null ? number + " " : "";
    s += postalCode != null ? postalCode + " " : "";
    s += city != null ? city + " " : "";
    s += country != null ? country + " " : "";

    return s;
  }

}
```

Add the JFace Data Binding plug-ins as dependency to your plug-in.

Change the `View` class to the following.

```
package de.vogella.databinding.example;

import org.eclipse.core.databinding.Binding;
import org.eclipse.core.databinding.DataBindingContext;
```

```
import org.eclipse.core.databinding.DataBindingContext;
import org.eclipse.core.databinding.UpdateValueStrategy;
import org.eclipse.core.databinding.beans.BeanProperties;
import org.eclipse.core.databinding.observable.value.IObservableValue;
import org.eclipse.core.databinding.validation.IValidator;
import org.eclipse.core.databinding.validation.ValidationStatus;
import org.eclipse.core.runtime.IStatus;
import org.eclipse.jface.databinding.fieldassist.ControlDecorationSupport;
import org.eclipse.jface.databinding.swt.WidgetProperties;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Combo;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;
import org.eclipse.ui.part.ViewPart;

import de.vogella.databinding.example.model.Address;
import de.vogella.databinding.example.model.Person;

public class View extends ViewPart {
  public static final String ID = "de.vogella.databinding.person.swt.View";
  private Person person;

  private Text firstName;
  private Text ageText;
  private Button marriedButton;
  private Combo genderCombo;
  private Text countryText;

  @Override
  public void createPartControl(Composite parent) {

    person = createPerson();
    // Lets put thing to order
    GridLayout layout = new GridLayout(2, false);
    layout.marginRight = 5;
    parent.setLayout(layout);

    Label firstLabel = new Label(parent, SWT.NONE);
    firstLabel.setText("Firstname: ");
    firstName = new Text(parent, SWT.BORDER);

    GridData gridData = new GridData();
    gridData.horizontalAlignment = SWT.FILL;
    gridData.grabExcessHorizontalSpace = true;
    firstName.setLayoutData(gridData);

    Label ageLabel = new Label(parent, SWT.NONE);
    ageLabel.setText("Age: ");
    ageText = new Text(parent, SWT.BORDER);

    gridData = new GridData();
```

```
gridData.horizontalAlignment = SWT.FILL;
gridData.grabExcessHorizontalSpace = true;
ageText.setLayoutData(gridData);

Label marriedLabel = new Label(parent, SWT.NONE);
marriedLabel.setText("Married: ");
marriedButton = new Button(parent, SWT.CHECK);

Label genderLabel = new Label(parent, SWT.NONE);
genderLabel.setText("Gender: ");
genderCombo = new Combo(parent, SWT.NONE);
genderCombo.add("Male");
genderCombo.add("Female");

Label countryLabel = new Label(parent, SWT.NONE);
countryLabel.setText("Country");
countryText = new Text(parent, SWT.BORDER);

Button button1 = new Button(parent, SWT.PUSH);
button1.setText("Write model");
button1.addSelectionListener(new SelectionAdapter() {

  @Override
  public void widgetSelected(SelectionEvent e) {
    System.out.println("Firstname: " + person.getFirstName());
    System.out.println("Age " + person.getAge());
    System.out.println("Married: " + person.isMarried());
    System.out.println("Gender: " + person.getGender());
    System.out.println("Country: "
        + person.getAddress().getCountry());
  }
});

Button button2 = new Button(parent, SWT.PUSH);
button2.setText("Change model");
button2.addSelectionListener(new SelectionAdapter() {
  @Override
  public void widgetSelected(SelectionEvent e) {
    person.setFirstName("Lars");
    person.setAge(person.getAge() + 1);
    person.setMarried(!person.isMarried());
    if (person.getGender().equals("Male")) {
      person.setGender("Male");
    } else {
      person.setGender("Female");
    }
    if (person.getAddress().getCountry().equals("Deutschland")) {
      person.getAddress().setCountry("USA");
    } else {
      person.getAddress().setCountry("Deutschland");
    }
  }
});

// now lets do the binding
bindValues();
```

```
  }

  private Person createPerson() {
    Person person = new Person();
    Address address = new Address();
    address.setCountry("Deutschland");
    person.setAddress(address);
    person.setFirstName("John");
    person.setLastName("Doo");
    person.setGender("Male");
    person.setAge(12);
    person.setMarried(true);
    return person;
  }

  @Override
  public void setFocus() {
  }

  private void bindValues() {
    // The DataBindingContext object will manage the databindings
    // Lets bind it
    DataBindingContext ctx = new DataBindingContext();
    IObservableValue widgetValue = WidgetProperties.text(SWT.Modify)
        .observe(firstName);
    IObservableValue modelValue = BeanProperties.value(Person.class,
        "firstName").observe(person);
    ctx.bindValue(widgetValue, modelValue);

    // Bind the age including a validator
    widgetValue = WidgetProperties.text(SWT.Modify).observe(ageText);
    modelValue = BeanProperties.value(Person.class, "age").observe(person);
    // add an validator so that age can only be a number
    IValidator validator = new IValidator() {
      @Override
      public IStatus validate(Object value) {
        if (value instanceof Integer) {
          String s = String.valueOf(value);
          if (s.matches("\\d*")) {
            return ValidationStatus.ok();
          }
        }
        return ValidationStatus.error("Not a number");
      }
    };

    UpdateValueStrategy strategy = new UpdateValueStrategy();
    strategy.setBeforeSetValidator(validator);

    Binding bindValue = ctx.bindValue(widgetValue, modelValue, strategy,
        null);
    // add some decorations
    ControlDecorationSupport.create(bindValue, SWT.TOP | SWT.LEFT);

    widgetValue = WidgetProperties.selection().observe(marriedButton);
    modelValue = BeanProperties.value(Person.class, "married").observe(person);
    ctx.bindValue(widgetValue, modelValue);
```

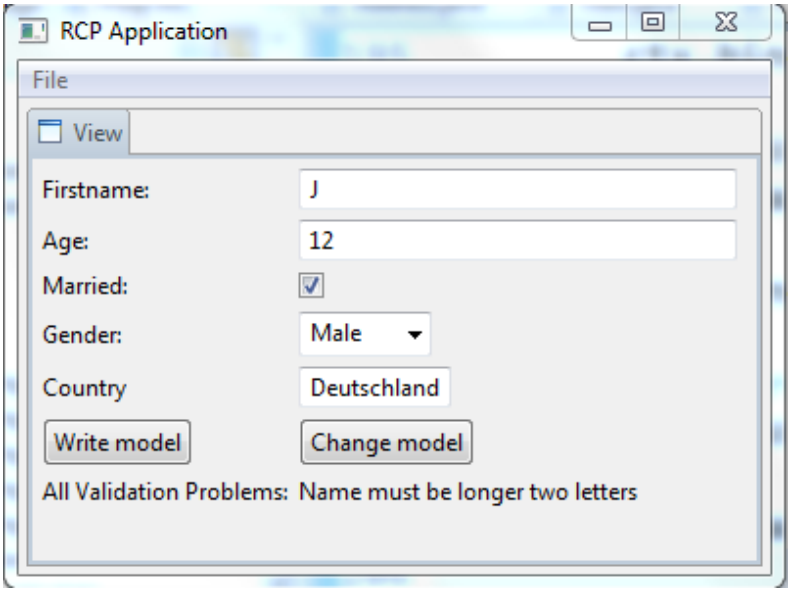
```
    ctx.bindValue(widgetValue, modelValue);

    widgetValue = WidgetProperties.selection().observe(genderCombo);
    modelValue = BeanProperties.value("gender").observe(person);

    ctx.bindValue(widgetValue, modelValue);

    // address field is bound to the Ui
    widgetValue = WidgetProperties.text(SWT.Modify).observe(countryText);

    modelValue = BeanProperties.value(Person.class, "address.country")
        .observe(person);
    ctx.bindValue(widgetValue, modelValue);

  }
}
```

Run the example and test it. Each time you change the UI element then model changes automatically. If you change the model then the UI will also update. Try to input something else then a number iN the age field you will get an error symbol in the UI and if the mouse hovers over the symbol you see the error message.

7.2. More Customer Validations and ControlDecoration

The following extends the example with the usage of *Validators* and *Decorators*.

In this example the *Validators* ensures that the firstName has at least 2 characters. A new label displays the validation status via a *Decorator*.



Create the following `StringLongerThenTwo` class.

```
package de.vogella.databinding.example.validators;

import org.eclipse.core.databinding.validation.IValidator;
import org.eclipse.core.databinding.validation.ValidationStatus;
import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.Status;

public class StringLongerThenTwo implements IValidator {

  @Override
  public IStatus validate(Object value) {
    if (value instanceof String) {
      String s = (String) value;
      // We check if the string is longer then 2 signs
      if (s.length() > 2) {
        return Status.OK_STATUS;
      } else {
        return ValidationStatus
            .error("Name must be longer two letters");
      }
    } else {
      throw new RuntimeException("Not supposed to be called for non-strings.");
    }
  }
}
```

The following shows the new code for `View.java`.

```
package de.vogella.databinding.example;

import org.eclipse.core.databinding.AggregateValidationStatus;
import org.eclipse.core.databinding.Binding;
import org.eclipse.core.databinding.DataBindingContext;
import org.eclipse.core.databinding.UpdateValueStrategy;
import org.eclipse.core.databinding.beans.BeanProperties;
import org.eclipse.core.databinding.beans.BeansObservables;
import org.eclipse.core.databinding.observable.value.IObservableValue;
import org.eclipse.core.databinding.validation.IValidator;
import org.eclipse.core.databinding.validation.ValidationStatus;
import org.eclipse.core.runtime.IStatus;
import org.eclipse.jface.databinding.fieldassist.ControlDecorationSupport;
import org.eclipse.jface.databinding.swt.WidgetProperties;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Combo;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;
import org.eclipse.ui.part.ViewPart;

import de.vogella.databinding.example.model.Address;
```

```
import de.vogella.databinding.example.model.Address;
import de.vogella.databinding.example.model.Person;
import de.vogella.databinding.example.validators.StringLongerThenTwo;

public class View extends ViewPart {
  public View() {
  }

  public static final String ID = "de.vogella.databinding.person.swt.View";
  private Person person;

  private Text firstName;
  private Text ageText;
  private Button marriedButton;
  private Combo genderCombo;
  private Text countryText;
  private Label errorLabel;

  @Override
  public void createPartControl(Composite parent) {

    person = createPerson();
    GridLayout layout = new GridLayout(2, false);
    layout.marginRight = 5;
    parent.setLayout(layout);

    Label firstLabel = new Label(parent, SWT.NONE);
    firstLabel.setText("Firstname: ");
    firstName = new Text(parent, SWT.BORDER);

    GridData gridData = new GridData();
    gridData.horizontalAlignment = SWT.FILL;
    gridData.grabExcessHorizontalSpace = true;
    firstName.setLayoutData(gridData);

    Label ageLabel = new Label(parent, SWT.NONE);
    ageLabel.setText("Age: ");
    ageText = new Text(parent, SWT.BORDER);

    gridData = new GridData();
    gridData.horizontalAlignment = SWT.FILL;
    gridData.grabExcessHorizontalSpace = true;
    ageText.setLayoutData(gridData);

    Label marriedLabel = new Label(parent, SWT.NONE);
    marriedLabel.setText("Married: ");
    marriedButton = new Button(parent, SWT.CHECK);

    Label genderLabel = new Label(parent, SWT.NONE);
    genderLabel.setText("Gender: ");
    genderCombo = new Combo(parent, SWT.NONE);
    genderCombo.add("Male");
    genderCombo.add("Female");

    Label countryLabel = new Label(parent, SWT.NONE);
    countryLabel.setText("Country");
    countryText = new Text(parent, SWT.BORDER);
```

```
    Button button1 = new Button(parent, SWT.PUSH);
    button1.setText("Write model");
    button1.addSelectionListener(new SelectionAdapter() {

      @Override
      public void widgetSelected(SelectionEvent e) {
        System.out.println("Firstname: " + person.getFirstName());
        System.out.println("Age " + person.getAge());
        System.out.println("Married: " + person.isMarried());
        System.out.println("Gender: " + person.getGender());
        System.out.println("Country: "
            + person.getAddress().getCountry());
      }
    });

    Button button2 = new Button(parent, SWT.PUSH);
    button2.setText("Change model");
    button2.addSelectionListener(new SelectionAdapter() {
      @Override
      public void widgetSelected(SelectionEvent e) {
        person.setFirstName("Lars");
        person.setAge(person.getAge() + 1);
        person.setMarried(!person.isMarried());
        if (person.getGender().equals("Male")) {

        } else {
          person.setGender("Male");
        }
        if (person.getAddress().getCountry().equals("Deutschland")) {
          person.getAddress().setCountry("USA");
        } else {
          person.getAddress().setCountry("Deutschland");
        }
      }
    });

    // this label displays all errors of all bindings
    Label descAllLabel = new Label(parent, SWT.NONE);
    descAllLabel.setText("All Validation Problems:");
    errorLabel = new Label(parent, SWT.NONE);
    gridData = new GridData();
    gridData.horizontalAlignment = SWT.FILL;
    gridData.grabExcessHorizontalSpace = true;
    gridData.horizontalAlignment = GridData.FILL;
    gridData.horizontalSpan = 1;
    errorLabel.setLayoutData(gridData);

    // perform the binding
    bindValues();
  }

  private Person createPerson() {
    Person person = new Person();
    Address address = new Address();
    address.setCountry("Deutschland");
```

```
    person.setAddress(address);
    person.setFirstName("John");
    person.setLastName("Doo");
    person.setGender("Male");
    person.setAge(12);
    person.setMarried(true);
    return person;
  }

  @Override
  public void setFocus() {
  }

  private void bindValues() {
    // the DataBindingContext object will manage the databindings
    DataBindingContext ctx = new DataBindingContext();
    IObservableValue widgetValue = WidgetProperties.text(SWT.Modify)
        .observe(firstName);
    IObservableValue modelValue = BeanProperties.value(Person.class,
        "firstName").observe(person);
    // define the UpdateValueStrategy
    UpdateValueStrategy update = new UpdateValueStrategy();
    update.setAfterConvertValidator(new StringLongerThenTwo());
    ctx.bindValue(widgetValue, modelValue, update, null);

    // bind the age including a validator
    widgetValue = WidgetProperties.text(SWT.Modify).observe(ageText);
    modelValue = BeanProperties.value(Person.class, "age").observe(person);
    // add an validator so that age can only be a number
    IValidator validator = new IValidator() {
      @Override
      public IStatus validate(Object value) {
        if (value instanceof Integer) {
          String s = String.valueOf(value);
          if (s.matches("\\d*")) {
            return ValidationStatus.ok();
          }
        }
        return ValidationStatus.error("Not a number");
      }
    };

    UpdateValueStrategy strategy = new UpdateValueStrategy();
    strategy.setBeforeSetValidator(validator);

    Binding bindValue = ctx.bindValue(widgetValue, modelValue, strategy,
        null);
    // add some decorations
    ControlDecorationSupport.create(bindValue, SWT.TOP | SWT.LEFT);

    widgetValue = WidgetProperties.selection().observe(marriedButton);
    modelValue = BeanProperties.value(Person.class, "married").observe(person);
    ctx.bindValue(widgetValue, modelValue);

    widgetValue = WidgetProperties.selection().observe(genderCombo);
    modelValue = BeanProperties.value("gender").observe(person)
```

```
    ctx.bindValue(widgetValue, modelValue);

    widgetValue = WidgetProperties.text(SWT.Modify).observe(countryText);

    modelValue = BeanProperties.value(Person.class, "address.country")
        .observe(person);
    ctx.bindValue(widgetValue, modelValue);

    // listen to all errors via this binding
    // we do not need to listen to any SWT event on this label as it never
    // changes independently
    final IObservableValue errorObservable = WidgetProperties.text()
        .observe(errorLabel);
    // this one listenes to all changes
    ctx.bindValue(errorObservable,
        new AggregateValidationStatus(ctx.getBindings(),
            AggregateValidationStatus.MAX_SEVERITY), null, null);

  }
}
```

8. Tutorial: WritableValue

Create a new View in your "de.vogella.databinding.example" plug-in with the following class. Via the buttons you can change the details of the WritableObject.

```
package de.vogella.databinding.example;

import org.eclipse.core.databinding.DataBindingContext;
import org.eclipse.core.databinding.beans.BeanProperties;
import org.eclipse.core.databinding.observable.value.IObservableValue;
import org.eclipse.core.databinding.observable.value.WritableValue;
import org.eclipse.jface.databinding.swt.WidgetProperties;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Text;

import de.vogella.databinding.example.model.Person;

public class ViewWritableValue extends View {
  private WritableValue value;

  @Override
  public void createPartControl(Composite parent) {
    value = new WritableValue();
    parent.setLayout(new GridLayout(3, false));
    GridData gd = new GridData();
    gd.grabExcessHorizontalSpace = true;
    Text text = new Text(parent, SWT.BORDER);
```

```
    Button button = new Button(parent, SWT.PUSH);
    button.setText("New Person");
    button.addSelectionListener(new SelectionAdapter() {
      @Override
      public void widgetSelected(SelectionEvent e) {
        Person p = new Person();
        p.setFirstName("Lars");
        value.setValue(p);
      }
    });

    button = new Button(parent, SWT.PUSH);
    button.setText("Another Person");
    button.addSelectionListener(new SelectionAdapter() {
      @Override
      public void widgetSelected(SelectionEvent e) {
        Person p = new Person();
        p.setFirstName("Jack");
        value.setValue(p);
      }
    });
    DataBindingContext ctx = new DataBindingContext();
    IObservableValue target = WidgetProperties.text(SWT.Modify).observe(text);
    IObservableValue model = BeanProperties.value("firstName")
        .observeDetail(value);
    ctx.bindValue(target, model);
  }

  @Override
  public void setFocus() {
  }
}
```

9. Tutorial: Data Binding for a JFace Viewer

Create a new Eclipse RCP project "de.vogella.databinding.viewer" using the "RCP Application with a view" template. Add the databinding plug-ins as dependency to your plug-in project.

Create the `de.vogella.databinding.viewer.model` package and re-create the `Person` and `Address` class from the previous example in this book in this package.

Create the following `MyModel` class to get some example data.

```java
package de.vogella.databinding.viewer.model;

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

public class MyModel {
  public static List<Person> getPersons() {
    List<Person> persons = new ArrayList<Person>();
    Person p = new Person();
    p.setFirstName("Joe");
    p.setLastName("Darcey");
    persons.add(p);
    p = new Person();
    p.setFirstName("Jim");
    p.setLastName("Knopf");
    persons.add(p);
    p = new Person();
    p.setFirstName("Jim");
    p.setLastName("Bean");
    persons.add(p);
    return persons;
  }
}
```

Create a new view called *ViewTable* add it to your RCP application. Change ViewTable.java to the following.

```java
package de.vogella.databinding.viewer;

import org.eclipse.core.databinding.beans.BeanProperties;
import org.eclipse.core.databinding.observable.list.WritableList;
import org.eclipse.jface.databinding.viewers.ViewerSupport;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.jface.viewers.TableViewer;
import org.eclipse.jface.viewers.TableViewerColumn;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;

import de.vogella.databinding.viewer.model.MyModel;
import de.vogella.databinding.viewer.model.Person;

public class ViewTable extends View {
  private TableViewer viewer;
  private WritableList input;

  @Override
  public void createPartControl(Composite parent) {
    parent.setLayout(new GridLayout(1, false));
    GridData gd = new GridData();
    gd.grabExcessHorizontalSpace = true;
```

```
// Define the viewer
viewer = new TableViewer(parent);
viewer.getControl().setLayoutData(new GridData(SWT.FILL, SWT.FILL, true, true));
TableViewerColumn column = new TableViewerColumn(viewer, SWT.NONE);
column.getColumn().setWidth(100);
column.getColumn().setText("First Name");
column = new TableViewerColumn(viewer, SWT.NONE);
column.getColumn().setWidth(100);
column.getColumn().setText("Last Name");
column = new TableViewerColumn(viewer, SWT.NONE);
column.getColumn().setWidth(100);
column.getColumn().setText("Married");
viewer.getTable().setHeaderVisible(true);

// now lets bind the values
// No extra label provider / content provider / setInput required
input = new WritableList(MyModel.getPersons(), Person.class);
ViewerSupport.bind(viewer,
    input,
    BeanProperties.values(new String[] { "firstName", "lastName",
        "married" }));

// The following buttons are there to test the binding
Button delete = new Button(parent, SWT.PUSH);
delete.setText("Delete");
delete.addSelectionListener(new SelectionAdapter() {
  @Override
  public void widgetSelected(SelectionEvent e) {
    if (!viewer.getSelection().isEmpty()) {
      IStructuredSelection selection = (IStructuredSelection) viewer
          .getSelection();
      Person p = (Person) selection.getFirstElement();
      input.remove(p);
    }
  }
});

Button add = new Button(parent, SWT.PUSH);
add.setText("Add");
add.addSelectionListener(new SelectionAdapter() {
  @Override
  public void widgetSelected(SelectionEvent e) {
    Person p = new Person();
    p.setFirstName("Test1");
    p.setLastName("Test2");
    input.add(p);
  }
});
Button change = new Button(parent, SWT.PUSH);
change.setText("Switch First / Lastname");
change.addSelectionListener(new SelectionAdapter() {
  @Override
  public void widgetSelected(SelectionEvent e) {
    if (!viewer.getSelection().isEmpty()) {
      IStructuredSelection selection = (IStructuredSelection) viewer
          .getSelection();
```

```
            Person p = (Person) selection.getFirstElement();
            String temp = p.getLastName();
            p.setLastName(p.getFirstName());
            p.setFirstName(temp);
          }
        }
      });

    }

    @Override
    public void setFocus() {
      viewer.getControl().setFocus();
    }
  }
```

In this example the user interface is updated if you delete and element or add an element to the collection. Run this example and test it.

10. Using ObservableListContentProvider and ObservableMapLabelProvider

If you use `WritableList` and `ObservableListContentProvider` you only listens to the changes in the list. You can use `ObservableMapLabelProvider` to listen to changes of the individual objects.

Change the `View.java` to the following.

```
package de.vogella.databinding.viewer;

import java.util.List;

import org.eclipse.core.databinding.beans.BeanProperties;
import org.eclipse.core.databinding.observable.list.WritableList;
import org.eclipse.core.databinding.observable.map.IObservableMap;
import org.eclipse.core.databinding.observable.set.IObservableSet;
import org.eclipse.jface.databinding.viewers.ObservableListContentProvider;
import org.eclipse.jface.databinding.viewers.ObservableMapLabelProvider;
import org.eclipse.jface.viewers.ILabelProvider;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.jface.viewers.ListViewer;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.ui.part.ViewPart;

import de.vogella.databinding.viewer.model.MyModel;
import de.vogella.databinding.viewer.model.Person;

// direct usage of ObservableListContentProvider
// listens to the labels changes too via ObservableMapLabelProvider

public class View extends ViewPart {
```

```
  private ListViewer viewer;
  private WritableList input;

  @Override
  public void createPartControl(Composite parent) {
    parent.setLayout(new GridLayout(1, false));
    GridData gd = new GridData();
    gd.grabExcessHorizontalSpace = true;

    // define the viewer
    viewer = new ListViewer(parent);
    viewer.getControl().setLayoutData(new GridData(SWT.FILL, SWT.FILL, true, true));
    ObservableListContentProvider contentProvider = new ObservableListContentProvider();
    viewer.setContentProvider(contentProvider);

    // create the label provider including monitoring
    // of label changes
    IObservableSet knownElements = contentProvider.getKnownElements();
    final IObservableMap firstNames = BeanProperties.value(Person.class,
        "firstName").observeDetail(knownElements);
    final IObservableMap lastNames = BeanProperties.value(Person.class,
        "lastName").observeDetail(knownElements);

    IObservableMap[] labelMaps = { firstNames, lastNames };

    ILabelProvider labelProvider = new ObservableMapLabelProvider(labelMaps) {
      public String getText(Object element) {
        return firstNames.get(element) + " " + lastNames.get(element);
      }
    };

    viewer.setLabelProvider(labelProvider);

    // create sample data
    List<Person> persons = MyModel.getPersons();
    input = new WritableList(persons, Person.class);
    // set the writeableList as input for the viewer
    viewer.setInput(input);

    Button delete = new Button(parent, SWT.PUSH);
    delete.setText("Delete");
    delete.addSelectionListener(new SelectionAdapter() {
      @Override
      public void widgetSelected(SelectionEvent e) {
        deletePerson();
      }

    });

    Button add = new Button(parent, SWT.PUSH);
    add.setText("Add");
    add.addSelectionListener(new SelectionAdapter() {
      @Override
      public void widgetSelected(SelectionEvent e) {
        addPerson();
      }
```

```
    });
    Button change = new Button(parent, SWT.PUSH);
    change.setText("Switch First / Lastname");
    change.addSelectionListener(new SelectionAdapter() {
      @Override
      public void widgetSelected(SelectionEvent e) {
        switchFirstLastName();
      }

    });
  }

  public void switchFirstLastName() {
    if (!viewer.getSelection().isEmpty()) {
      IStructuredSelection selection = (IStructuredSelection) viewer
          .getSelection();
      Person p = (Person) selection.getFirstElement();
      String temp = p.getLastName();
      p.setLastName(p.getFirstName());
      p.setFirstName(temp);
    }
  }

  public void deletePerson() {
    if (!viewer.getSelection().isEmpty()) {
      IStructuredSelection selection = (IStructuredSelection) viewer
          .getSelection();
      Person p = (Person) selection.getFirstElement();
      input.remove(p);
    }
  }

  public void addPerson() {
    Person p = new Person();
    p.setFirstName("Test1");
    p.setLastName("Test2");
    input.add(p);
  }

  @Override
  public void setFocus() {
    viewer.getControl().setFocus();
  }

}
```

11. About this website

11.1. Donate to support free tutorials

Please consider a contribution  if this article helped you. It will help to maintain our content and our Open Source activities.

11.2. Questions and discussion

Writing and updating these tutorials is a lot of work. If this free community service was helpful, you can support the cause by giving a tip as well as reporting typos and factual errors.

If you find errors in this tutorial, please notify me (see the **top of the page**). Please note that due to the high volume of feedback I receive, I cannot answer questions to your implementation. Ensure you have read the **vogella FAQ** as I don't respond to questions already answered there.

11.3. License for this tutorial and its code

12. Links and Literature

12.1. Source Code

Source Code of Examples

12.2. Eclipse Data Binding resources

Wiki about the JFace Data Binding

WindowBuilder Data Binding Example

Using Bean Validation (JSR 303) with JFace Data Binding

12.3. vogella Resources

TRAINING

The vogella company provides comprehensive **training and education services** from experts in the areas of Eclipse RCP, Android, Git, Java, Gradle and Spring. We offer both public and inhouse training. Whichever course you decide to take, you are guaranteed to experience what many before you refer to as **"The best IT class I have ever attended"**.

SERVICE & SUPPORT

The vogella company offers **expert consulting** services, development support and coaching. Our customers range from Fortune 100 corporations to individual developers.