

Understanding JFace data binding in Eclipse, Part 2: The basics of binding

Skill Level: Intermediate

[Scott Delap \(scott@clientjava.com\)](mailto:scott@clientjava.com)
Desktop/Enterprise Java Consultant

03 Oct 2006

Almost all applications require synchronization of data between various objects and services. However, moving String instances and writing event notification code can be tedious. Desktop application development is especially full of such tasks. The new JFace data binding application programming interface (API) included in Eclipse V3.2 looks to make this data synchronization process far easier. This "[Understanding JFace data binding in Eclipse](#)" series introduces basic and advanced uses of the JFace data binding API.

Section 1. Before you start

About this series

Data binding APIs relieve you from having to write boilerplate synchronization code. The JFace data binding API provides this functionality for user interfaces (UIs) written in the Standard Widget Toolkit (SWT) and JFace.

[Part 1](#) of this "[Understanding JFace data binding in Eclipse](#)" series explains what a data binding framework does, introduces several popular Java GUI data binding frameworks, and covers the pros and cons of using data binding. This tutorial, [Part 2](#), introduces the basic API components. [Part 3](#) moves on to advanced topics, such as tables, converters, and validation.

About this tutorial

This tutorial explains reasons for using a data binding API. It then introduces you to using the core components of the JFace data binding API while laying the groundwork for more advanced topics, covered in Part 3.

Prerequisites

This tutorial is written for developers with some experience with the Java™ programming language and Eclipse. You should also have a basic understanding of SWT and JFace.

System requirements

To run the examples, you need a copy of the Eclipse software development kit (SDK) and a machine capable of running it.

Section 2. Synchronizing data between domain objects and controls

The need for synchronization

Desktop applications often feature long-lived objects that contain data visible to the user. A change in the first-name field in a person object, for instance, usually needs to be reflected in the form in which a user is editing that object. This means updating the text field widget that displays the data. If the change was initiated in the text field widget, the person object needs to be updated. If the person object changed as the result of business process, the widget displaying the change needs modification.

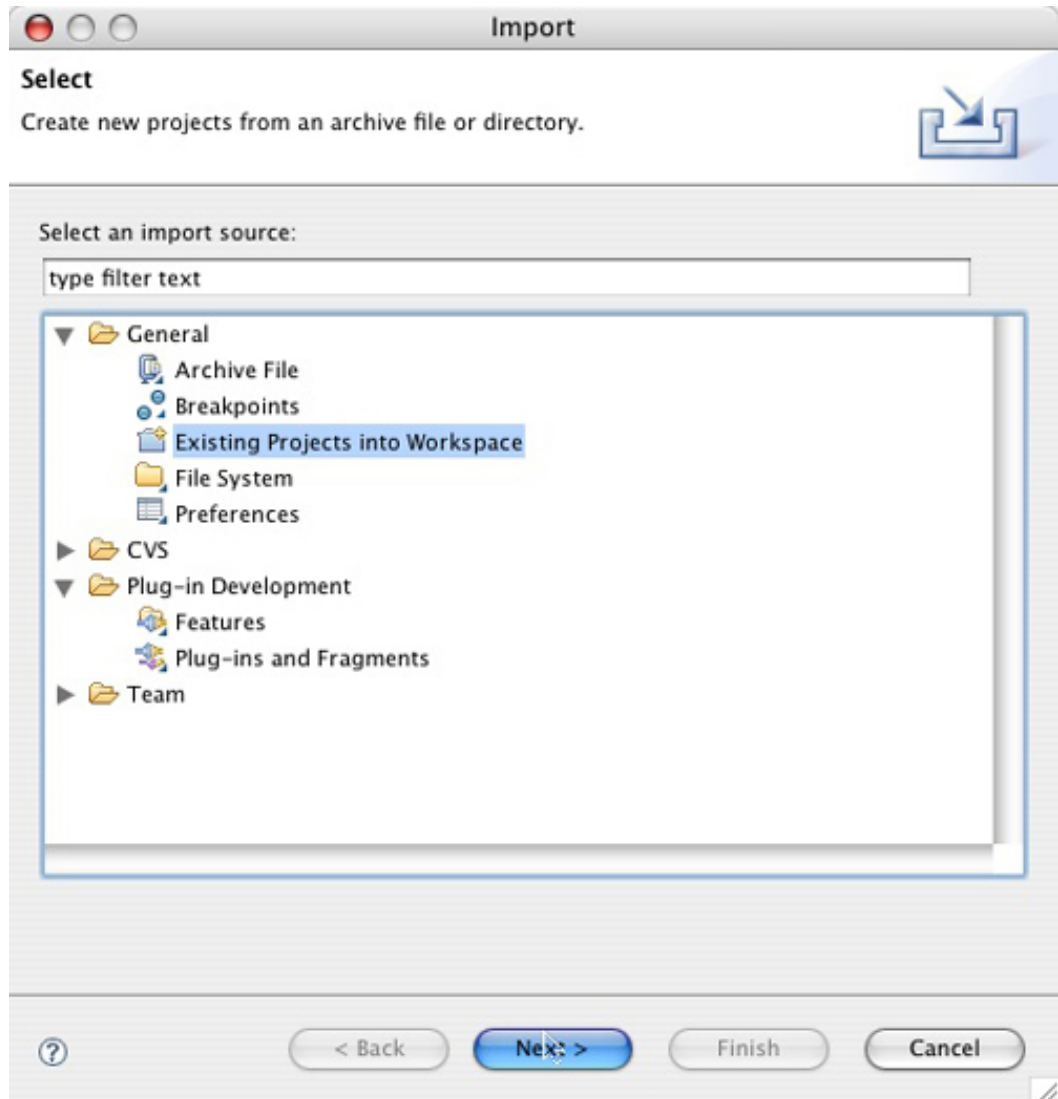
Many widgets, such as tables and lists, have models that make this process easier. Changing the model automatically notifies the widget. That being said, most application data isn't in the form of an SWT-specific model. In the case of a table, the data used to populate it often takes the form of a `java.util.List` of values queried from a server or database. Further complicating the situation is the fact that some widgets (like text fields) don't have models at all; they have simple properties inherent to the widget that contain the data that is displayed.

Boilerplate synchronization

The two primary Java widget toolkits, Swing and SWT, feature widgets that are not data-aware. This means it's up to you to manage the synchronization processes. It's helpful to look at an example. Follow these steps:

1. Open Eclipse V3.2 and create a new workspace.
2. Select **File > Import** from the menu. The Eclipse Project Import Wizard opens (see Figure 1).

Figure 1. The Eclipse Project Import Wizard



3. Choose **Existing Projects into Workspace** and click **Next**.
4. On the next screen, choose **Select archive file** and import the project.zip file available for download in the [Downloads](#) section of this tutorial (see

Figure 2). Your workspace should now contain a project that looks similar to the one shown in Figure 3.

Figure 2. Selecting the project archive

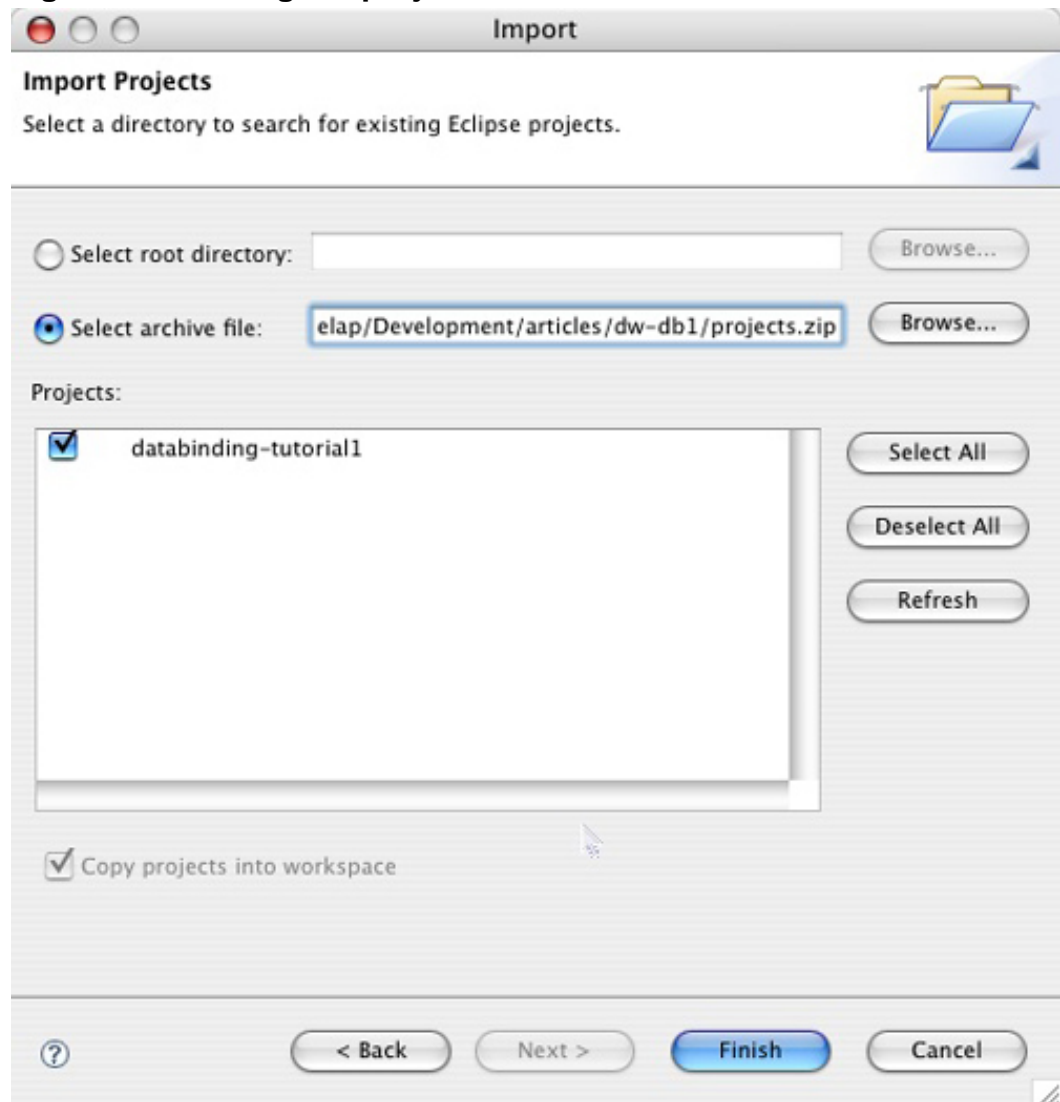
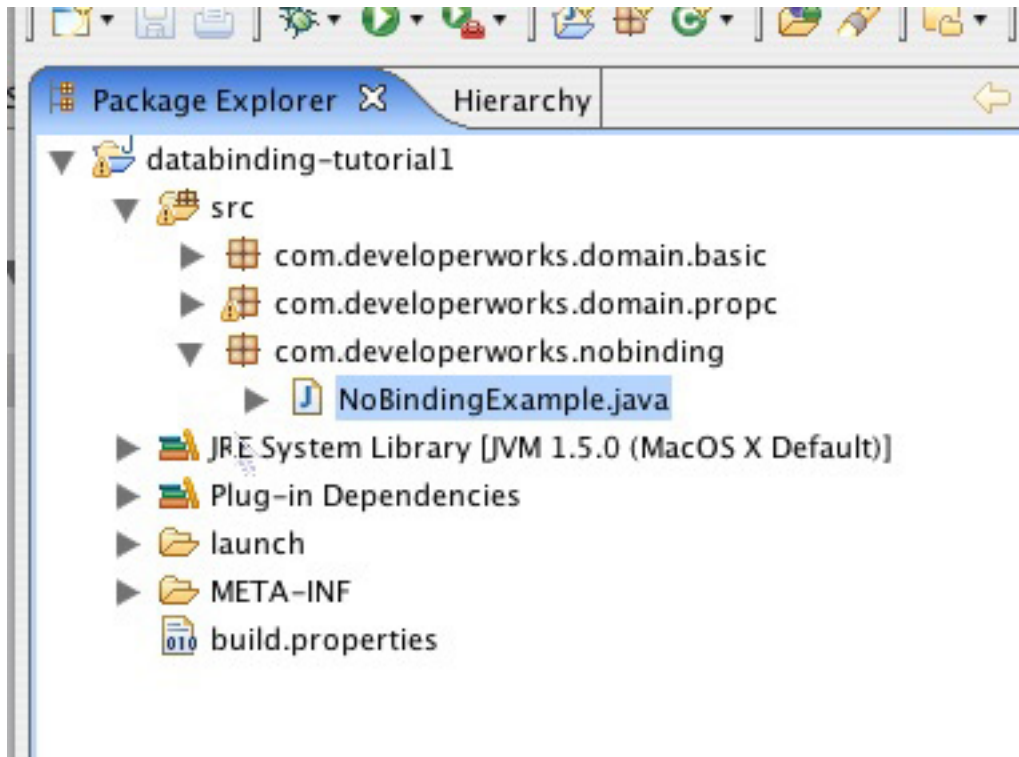
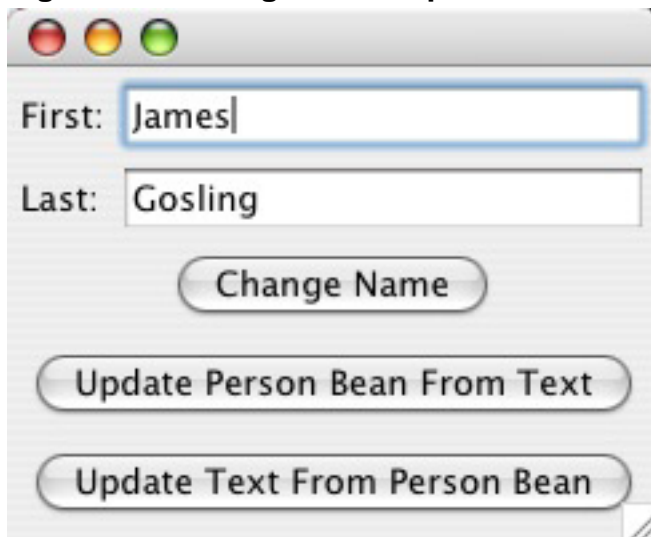


Figure 3. The workspace after project import



5. Click the arrow next to the Eclipse run icon and select the **NoBinding** run target. A window appears that looks similar to the one shown in Figure 4.
Figure 4. Running the example



At this point, it's useful to perform a few exercises with the application:

1. Notice that there is no text present in either text box. Click **Change Name** to change the text to James Gosling.

2. Change the First and Last name fields to anything of your choosing.
3. Click **Update Text From Person Bean**. The text reverts to James Gosling. This happens because the field changes you made were not synchronized with the `Person` bean.
4. Change the text again and click **Update Person Bean From Text**.
5. Change the text again and click **Update Text from Person Bean**. The text changes back to the text you first entered because these values are manually synchronized with the `Person` bean when you click the button.

The code for this example is shown below.

Listing 1. An example application with manual synchronization

```
public class Person {
    private String first;
    private String last;

    public Person(String first, String last) {
        this.first = first;
        this.last = last;
    }

    public String getFirst() {
        return first;
    }
    public void setFirst(String first) {
        this.first = first;
    }
    public String getLast() {
        return last;
    }
    public void setLast(String last) {
        this.last = last;
    }
}

public class NoBindingExample {
    private Person person;
    private Text firstText;
    private Text lastText;

    private void createControls(Shell shell) {
        GridLayout gridLayout = new GridLayout();
        gridLayout.numColumns = 2;
        shell.setLayout(gridLayout);

        Label label = new Label(shell, SWT.SHELL_TRIM);
        label.setText("First:");

        GridData gridData = new GridData(GridData.FILL_HORIZONTAL);
        this.firstText = new Text(shell, SWT.BORDER);
        this.firstText.setLayoutData(gridData);

        label = new Label(shell, SWT.NONE);
        label.setText("Last:");
        this.lastText = new Text(shell, SWT.BORDER);
    }
}
```

```

        gridData = new GridData(GridData.FILL_HORIZONTAL);
        this.lastText.setLayoutData(gridData);
    }

    private void createButtons(Shell shell) {
        GridData gridData;
        gridData = new GridData();
        gridData.horizontalAlignment = SWT.CENTER;
        gridData.horizontalSpan = 2;
        Button button = new Button(shell, SWT.PUSH);
        button.setLayoutData(gridData);
        button.setText("Change Name");

        button.addSelectionListener(new SelectionAdapter() {

            public void widgetSelected(SelectionEvent e) {
                updatePerson();
                synchronizePersonToUI();
            }

        });

        gridData = new GridData();
        gridData.horizontalAlignment = SWT.CENTER;
        gridData.horizontalSpan = 2;
        button = new Button(shell, SWT.PUSH);
        button.setLayoutData(gridData);
        button.setText("Update Person Bean From Text");

        button.addSelectionListener(new SelectionAdapter() {

            public void widgetSelected(SelectionEvent e) {
                synchronizeUIToPerson();
            }

        });

        gridData = new GridData();
        gridData.horizontalAlignment = SWT.CENTER;
        gridData.horizontalSpan = 2;
        button = new Button(shell, SWT.PUSH);
        button.setLayoutData(gridData);
        button.setText("Update Text From Person Bean");

        button.addSelectionListener(new SelectionAdapter() {

            public void widgetSelected(SelectionEvent e) {
                synchronizePersonToUI();
            }

        });
    }

    private void updatePerson() {
        person.setFirst("James");
        person.setLast("Gosling");
    }

    private void synchronizePersonToUI() {
        this.firstText.setText(this.person.getFirst());
        this.lastText.setText(this.person.getLast());
    }

    private void synchronizeUIToPerson() {
        this.person.setFirst(this.firstText.getText());
        this.person.setLast(this.lastText.getText());
    }

    public static void main(String[] args) {

```

```
        NoBindingExample example = new NoBindingExample();
        example.run();
    }

    public void run() {
        this.person = new Person("Larry", "Wall");

        Display display = new Display();
        Shell shell = new Shell(display);

        createControls(shell);
        createButtons(shell);

        shell.pack();
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch())
                display.sleep();
        }
        display.dispose();
    }
}
```

Looking at the code

At the beginning of Listing 1 is the definition of a simple `Person` class that adheres to the JavaBean specification. Specifically, it features getter and setter methods for each of its properties. The listing continues by defining the `NoBindingExample` class. The main method instantiates an instance of the class and immediately delegates to the `run()` method. The `run()` method is responsible for creating the UI and starts the appropriate SWT constructs needed to display the example.

The `run()` method first creates a `Shell`, then calls the `createControls()` method that constructs the UI widgets. Next, it calls the `createButtons()` method, which creates three buttons. Each button has a mouse listener, which calls a specific method on the example instance.

Issues with this design

Thousands of applications have been written with code similar to this. However, there are number of issues:

- The `Person` bean initially contains the value `Larry Wall`. The application doesn't display this value at first because the `Person` bean isn't synced with the text fields at startup.
- References to the two text fields must be kept for use by the two sync methods.
- Boilerplate synchronization code must be written.

- Determining when to synchronize the values between the `Person` bean and the text fields is a manual process.

Even if this example was an application without buttons to synchronize values back and forth between the `Person` bean and the text fields, you'd still have to analyze, code, and maintain the process of when to call the synchronization methods. The situation would be simpler if the text fields were a reflection of the `Person` bean, and an API kept everything in sync, freeing you to concentrate on more pressing requirements.

Section 3. The magic of data binding

Fortunately, the API desired in the previous section isn't a dream. A number of frameworks are available for use with the Java language that solve this problem. They're commonly classified under the term *data binding*. Data binding frameworks do exactly what their name implies: They bind data between two points, updating one side of the relationship when the other side changes. This is the type of functionality the current example needs.

Eclipse V3.2 includes a provisional version of a data binding API in the `org.eclipse.jface.databinding` plug-in, which you can use to develop SWT and JFace applications. Future versions of Eclipse may include different versions of the API because of enhancements and redesign. This doesn't limit the usefulness of the current API, which is stable and includes many features. The rest of this tutorial uses it to redesign the previous example.

Importing data binding

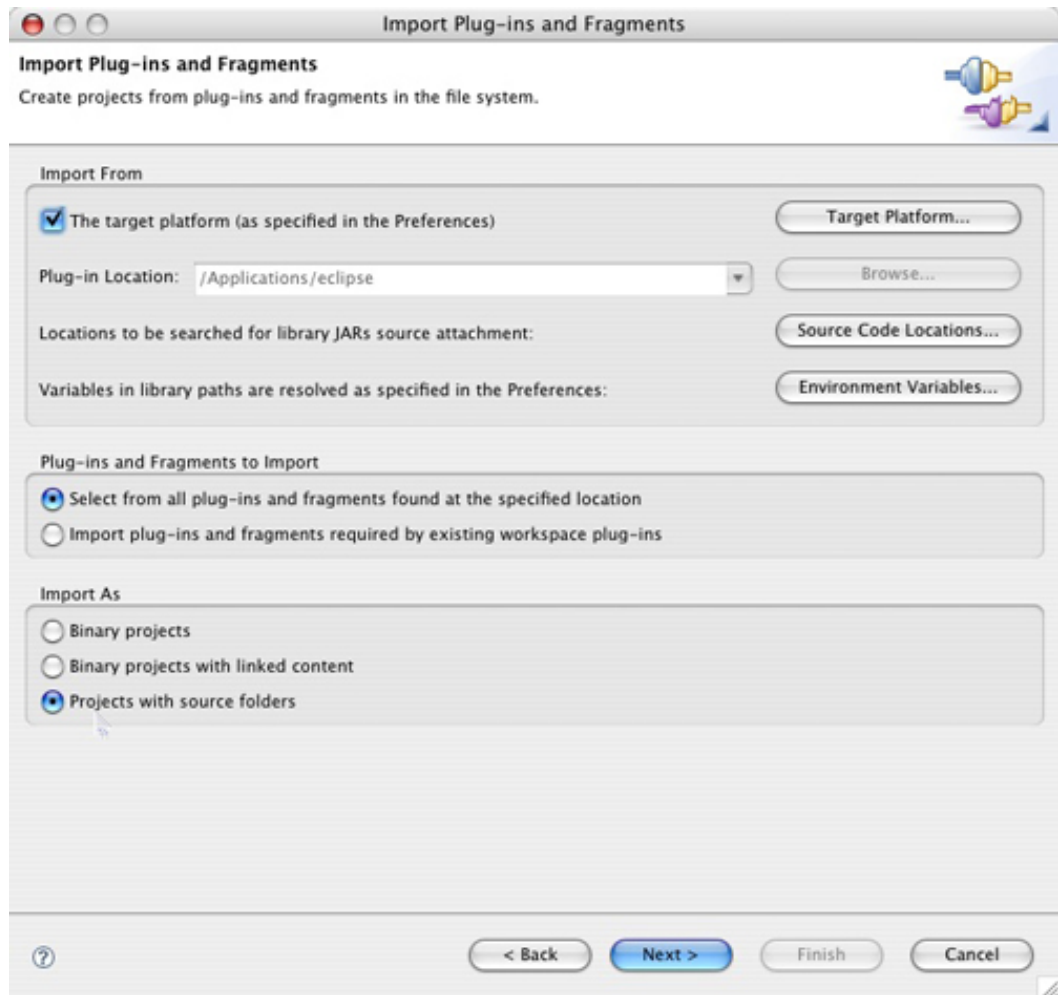
Although you could use the binary version of JFace data binding (the examples in the project archive run in your IDE now), it's useful to import the source as a reference during development. You can do this using the Eclipse Import Wizard, as follows:

1. Select **File > Import** from the menu.
2. Select **Plug-ins and Fragments**, as shown in Figure 5, and click **Next**.
Figure 5. Importing an existing plug-in

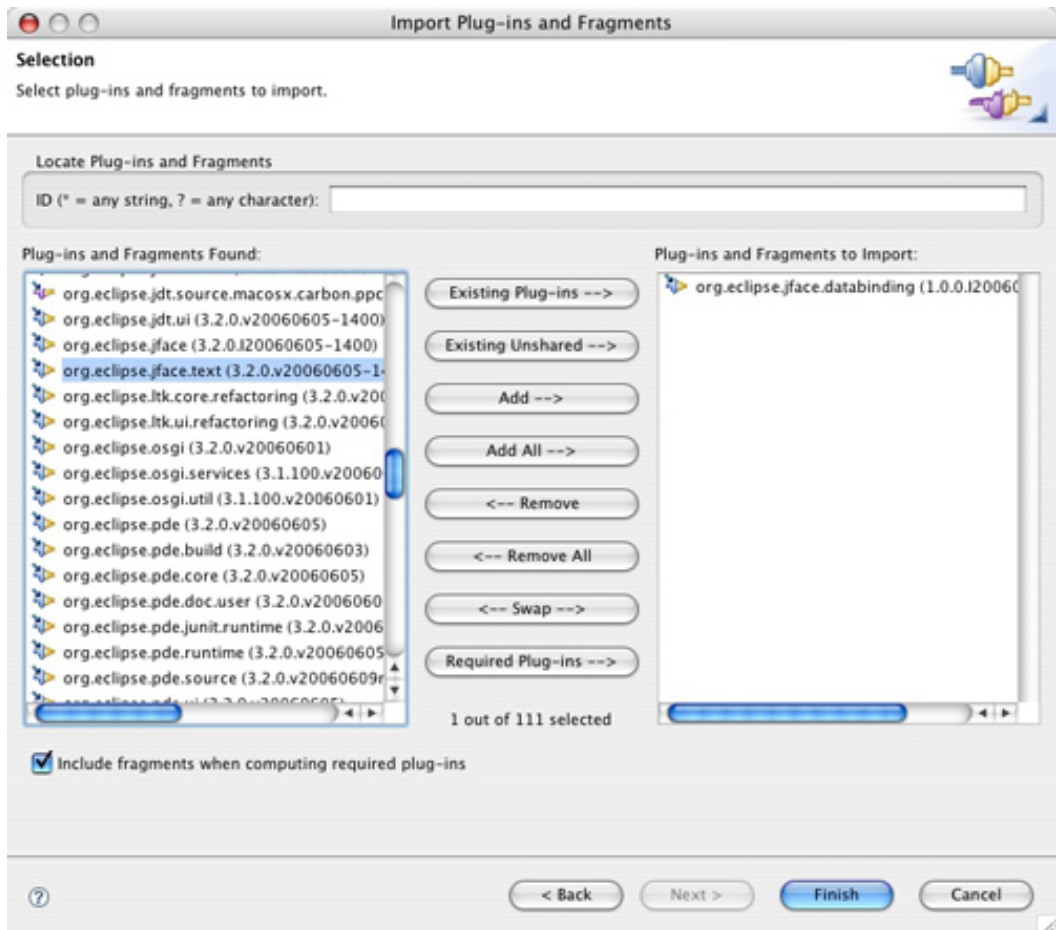


3. On the following screen, change the **Import As** option at the bottom to **Projects with Source Folders**, and click **Next** again, as shown in Figure 6.

Figure 6. Changing the Import As option to Source

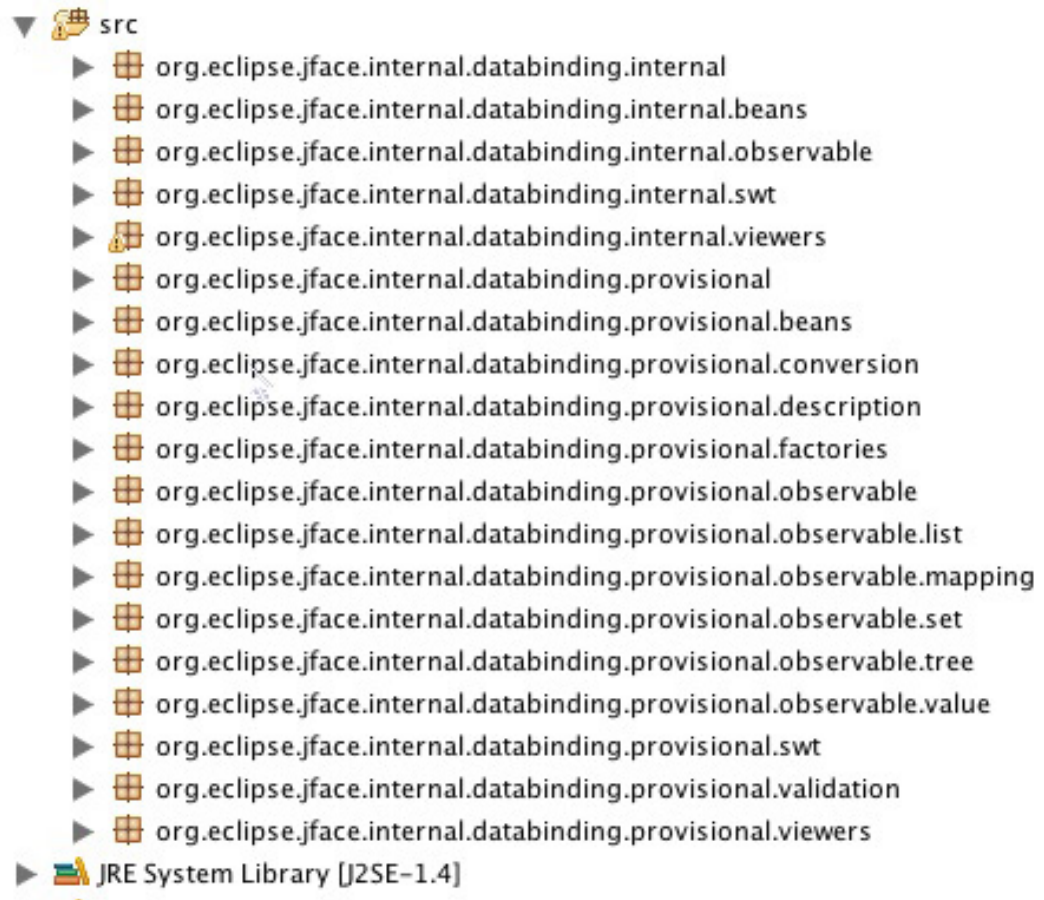


4. Select the **org.eclipse.jface.databinding** project from the list and move it to the right, as shown in Figure 7. Click **Finish** to import it.
Figure 7. Selecting the data binding plug-in



5. Expand the newly imported project. The resulting package listing is shown in Figure 8.

Figure 8. The workspace after import



Putting data binding to use

Instead of diving into the details of JFace data binding, you'll first put it to use, then learn how it works under the hood. Follow these steps:

1. Create a new package of your choosing in the databinding-tutorial project by right-clicking the src folder and selecting **New > Package** from the pop-up menu.
2. Copy the `NoBindingExample` class from the `com.developerworks.nobinding` package into the newly created package.
3. Rename the class to `BindingExample` by right-clicking it and choosing **Refactor > Rename**.
4. Paste the code from Listing 2 into the class before the `main()` method definition.

Listing 2. The createContext() method

```
public static DataBindingContext createContext() {
    DataBindingContext context =
        new DataBindingContext();
    context.addObservableFactory(
        new NestedObservableFactory(context));
    context.addObservableFactory(
        new BeanObservableFactory(
            context,
            null,
            new Class[] { Widget.class }));
    context.addObservableFactory(
        new SWTObservableFactory());
    context.addObservableFactory(
        new ViewersObservableFactory());
    context.addBindSupportFactory(
        new DefaultBindSupportFactory());
    context.addBindingFactory(
        new DefaultBindingFactory());
    context.addBindingFactory(
        new ViewersBindingFactory());
    return context;
}
```

5. Fix any imports as needed, then delete the `synchronizeUIToPerson()` method.
6. Delete the block of code that creates the **Update Person Bean From Text** button from the `createButtons()` method.
7. Paste the code from Listing 3 into the end of the `createControls()` method.

Listing 3. Binding Text Widget to the Person bean

```
DataBindingContext ctx = createContext();
ctx.bind(firstText,
        new Property(this.person, "first"),
        null);
ctx.bind(lastText,
        new Property(this.person, "last"),
        null);
```

8. Right-click your newly modified class and select **Run As > SWT Application** from the pop-up menu. You should see a window similar to Figure 9.

Figure 9. The modified example



9. Notice that the text widgets contain the initial values of `Larry` and `Wall`. Unlike the previous example, which forgot to synchronize the initial bean values, data binding has automatically handled this case. Type a few characters in the **First** field and click **Update Text From Person Bean**. The text reverts to its initial value.
10. Type a few characters again in the **First** field, but also tab into the **Last** field. Click **Update Text From Person Bean** again. The text you changed doesn't revert this time. Data binding automatically synchronized the value from the text widget into the `Person` bean's first `String` variable when focus was lost.

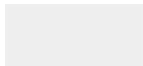
Section 4. How the magic works: Observables

You've now seen JFace data binding synchronizing data in a live application. The question remains: "How does it do that?"

The first step of any data binding framework is to abstract out the concepts of getting values, setting values, and listening to changes into general implementation. That general implementation can be used when referencing the concepts in the majority of the framework's code. Implementations for various cases can then be written to handle specific details.

JFace data binding abstracts out these concepts in the `IObservable` and `IObservableValue` interfaces, shown below.

Listing 4. The `IObservable` and `IObservableValue` interfaces



```
public
interface
IObservable
{
public
void
addChangeListener( IChangeListener
listener);
public
void
removeChangeListener( IChangeListener
listener);
public
void
addStaleListener( IStaleListener
listener);
public
void
removeStaleListener( IStaleListener
listener);
public
boolean
isStale();
public
void
dispose();
}

public
interface
IObservableValue
extends
IObservable
{

public
Object
getValueType();
public
Object
getValue();
public
void
setValue(Object
value);
public
void
addValueChangeListener( IValueChangeListener
listener);
public
void
removeValueChangeListener( IValueChangeListener
listener);
}
```

The `IObservable` interface defines a generic way to listen to change. The `IObservableValue` interface extends it to add the concept of a specific value, along with methods for getting and setting that value explicitly.

A way has now been defined to generically write code to handle any type of change for a specific value. All that is left is to adapt the things causing change -- the `Person` bean and the text widget -- to this interface.

Section 5. How the magic works: Observable factories

The `createContext()` method pasted into the `BindingExample` class contains the API that performs this adaptation process. JFace data binding makes the user of a series of observable factories that attempt to adapt an object into an observable when it's queried. If a factory can't adapt an object, null is returned, and the data-binding context tries the next factory in the list. If it's configured properly and the object type is supported, a suitable observable is returned. Listing 5 shows an excerpt from the `SWTObservableFactory` that generates observables for many common SWT controls.

The first case covered by the code excerpt's `if` block is that of a text widget. The `updatePolicy` property determines whether the `TextObservableValue` commits changes on modification (key pressed) or on focus lost. Notice that the `SWTObservableFactory` also supports other common SWT widgets, such as labels, combo boxes, lists, etc.

Listing 5. Factory code that builds a `TextObservable`

```
if
(description
instanceof
Text) {
int
updatePolicy
= new
int[] {
SWT.Modify,
SWT.FocusOut,
SWT.None
}[updateTime];
return
new
TextObservableValue\
((Text)
description,
updatePolicy);
} else
if
(description
instanceof
Button)
{
// int
updatePolicy
= new
int[] {
SWT.Modify,
SWT.FocusOut,
SWT.None
}[updateTime];
return
new
```

```
ButtonObservableValue((Button)
description);
} else
if
(description
instanceof
Label)
{
return
new
LabelObservableValue((Label)
description);
} else
if
(description
instanceof
Combo)
{
return
new
ComboObservableList((Combo)
description);
} else
if
(description
instanceof
Spinner)
{
return
new
SpinnerObservableValue((Spinner)
description,
SWTProperties.SELECTION);
} else
if
(description
instanceof
CCombo)
{
return
new
CComboObservableList((CCombo)
description);
} else
if
(description
instanceof
List) {
return
new
ListObservableList((List)
description);
}
```

Looking at the `TextObservableValue` class and its parents, shown in Listing 6, you can see that the getter and setter methods ultimately do call methods that adapt the text widget. The getter and setter map easily to the `getText()` and `setText()` methods. In addition, an update listener adapts focus changes into generic change events. It checks the update policy, specified when the `TextObservableValue` was created, and adapts the native text widget event into a generic `IObservableValue` event.

Listing 6. The get/set methods of the `TextObservableValue`

```

public final Object getValue() {
    ...
    return doGetValue();
}

public void setValue(Object value) {
    Object currentValue = doGetValue();
    ValueDiff diff = DiffUtils.createValueDiff(currentValue, value);

    ...

    doSetValue(value);
    fireValueChange(diff);
}

public void doSetValue(final Object value) {
    try {
        updating = true;
        bufferedValue = (String) value;
        text.setText(value == null ? "" : value.toString()); //$NON-NLS-1$
    } finally {
        updating = false;
    }
}

public Object doGetValue() {
    return text.getText();
}

private Listener updateListener = new Listener() {
    public void handleEvent(Event event) {
        if (!updating) {
            String oldValue = bufferedValue;
            String newValue = text.getText();
            // If we are updating on \
            // focus lost then when we fire the change
            // event change the buffered value
            if (updatePolicy == SWT.FocusOut) {
                bufferedValue = text.getText();

                if (!newValue.equals(oldValue)) {
                    fireValueChange\
                    (DiffUtils.createValueDiff(oldValue,
                                                newValue));
                }
            } else {
                fireValueChange\
                (DiffUtils.createValueDiff(oldValue, text
                                           .getText()));
            }
        }
    }
};

```

JFace data binding also supports adapting standard objects with JavaBean properties, such as the `Person` bean. The `BeanObservableFactory` adapts specific properties such as `first` in the example using the `JavaBeanObservable` object.

The `ctx.bind()` method calls you added when modifying the example put the observable factories to work. Code in the JFace data binding API takes the target and model objects, and searches for an appropriate observable adapter. Once one

has been found for each side of the binding relationship, they're wired together using an instance of the `ValueBinding()` class.

Section 6. How the magic works: ValueBinding

Once an observable has been created for the two entities to be bound, you need a third party that keeps them coordinated. This role is played by the `ValueBinding` class; a simplified excerpt is shown below.

Listing 7. An excerpt from ValueBinding

```
public
void
updateModelFromTarget()
{
    updateModelFromTarget\
(Diffs.createValueDiff(target.getValue(),
target
.getValue()));
}
public
void
updateModelFromTarget(ValueDiff
diff) {
    ...
    model.setValue(target.getValue());
    ...
}
```

An instance of `ValueBinding` listens to changes on the target- and model-generated observables, and synchronizes changes accordingly, using methods similar to Listing 7. As you can see, the `updateModelFromTarget()` method uses the generic access methods defined by the `IObservableValue` interface to retrieve the value from the target and set it on the model.

Section 7. How the magic works: Putting it all together

Step back to the `ctx.bind` lines in Listing 3 that were added to the `createControls()` method. Each `bind` method takes a target, a model, and a bind spec as arguments. (More details will be provided about bind specs in Part 2.)

Both the target and model must eventually be adapted to `IObservables` if the

target and models arguments don't implement the interface directly. In this example, this is done with the `IObservableFactory`. In the case of the `firstText` text widget, no other information is needed. It's assumed that when the target/model object specified to the `bind()` method is a text widget, the default binding should be made to its text property.

In the case of the `Person` bean, there is no obvious default property to bind to. As a result, the `Person` bean instance must be wrapped with a `Property` object. This object adds enough information with its `first` string to allow the `BeanObservableFactory` to determine which property on the `Person` bean to create an observable for.

With all this specification out of the way, the `bind()` method ultimately creates `IObservable` adapters for the target and model specified. It then creates an instance of `ValueBinding` to keep the values synchronized when one side of the relationship changes.

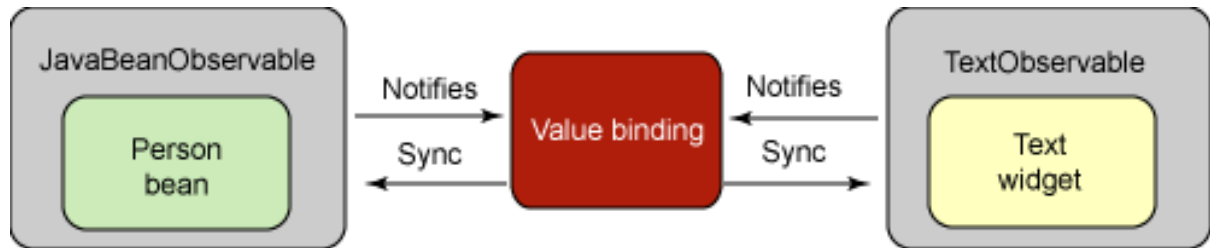
Now that the process has been described, it's useful to see the order of the method calls involved. Listing 8 shows a stacktrace from the time focus-loss occurs in the text widget until the breakpoint is hit in the `Person` bean because of JFace data binding synchronizing the value from the widget. Notice the various data binding and JavaBeans classes invoked in between -- you didn't have to write any of this code.

Listing 8. The stacktrace of a data binding synchronization

```
Text(Control).setTabItemFocus() line: 2958
Text(Control).forceFocus() line: 809
Text(Control).sendFocusEvent(int, boolean) line: 2290
Text(Widget).sendEvent(int) line: 1501
Text(Widget).sendEvent(int, Event, boolean) line: 1520
Text(Widget).sendEvent(Event) line: 1496
EventTable.sendEvent(Event) line: 66
TextObservableValue$1.handleEvent(Event) line: 51
TextObservableValue.access$5(TextObservableValue, ValueDiff) line: 1
TextObservableValue(AbstractObservableValue).fireValueChange(ValueDiff) line: 73
ValueBinding$2.handleValueChange(IObservableValue, ValueDiff) line: 135
ValueBinding.updateModelFromTarget(ValueDiff) line: 193
JavaBeanObservableValue.setValue(Object) line: 88
Method.invoke(Object, Object...) line: 585
DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: 25
NativeMethodAccessorImpl.invoke(Object, Object[]) line: 39
Person.setFirst(String) line: 17
```

Figure 10 shows a diagram of how the players in the synchronization process relate. Both the text widget and the `Person` bean are adapted to the `IObservableValue` interface. The `ValueBinding` class then listens to both adapters and uses them to synchronize changes to both sides of the relationship.

Figure 10. A diagram of the observable relationships



Section 8. Enabling change from your domain objects

If you go back to the `BindingExample` and look at the code in Listing 9, you'll notice that it still has a synchronization method to update the UI controls when the values in the `Person` bean changes. This is because the `Person` bean provides no notifications when its values change. This can easily be remedied by allowing JFace data binding to provide synchronization functionality.

Listing 9. The selection listener invoking synchronization from the `Person` bean to the UI

```

button.addSelectionListener(new
SelectionAdapter()
{
    public
    void \
    widgetSelected(SelectionEvent
    e) {
        updatePerson();
        synchronizePersonToUI();
    }
});

...

private
void
synchronizePersonToUI()
{
    this.firstText.setText(this.person.getFirst());
    this.lastText.setText(this.person.getLast());
}

```

Modify the `com.developerworks.basic.Person` class to extend the included `PropertyChangeAware` class. Then modify the two setter methods, as shown below.

Listing 10. Adding `PropertyChange` support to the setters

```

public void setFirst(String first) {

```

```
        Object oldVal = this.first;
        this.first = first;
        firePropertyChange("first", oldVal, this.first);
    }

    public String getLast() {
        return last;
    }

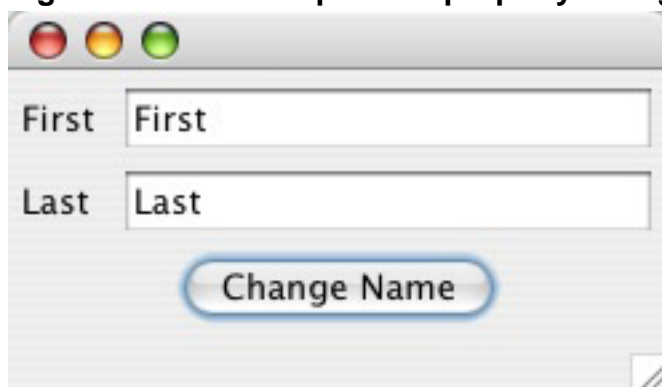
    public void setLast(String last) {
        Object oldVal = this.last;
        this.last = last;
        firePropertyChange("last", oldVal, this.last);
    }
}
```

The `PropertyChangeAware` class provides standard JavaBean property change support. Modifying the setters implements firing of `PropertyChangeEvents` when setters are called on the `Person` bean. The old value is stored, then the new value is set. Finally, a property change event with both values is fired for the specific property type. Notice that the property used follows the same naming convention as the setter method and adheres to the JavaBean specification. The `JavaBeanObservable` provided by JFace data binding supports listening to property change events of objects that it adapts. This allows it to convert specific property change events to more generic `IObservableValue` change events.

With these changes in place, remove the `syncPersonToUI()` method and the call to it in the Change Name button listener. Also remove the block of code that creates the Update Text From Person Bean button in the `createButtons()` method because it's no longer needed.

Starting the `BindingExample` shows a window similar to the one in Figure 11. Clicking **Change Name** results in the text widgets also changing. All the synchronization work is now done by JFace data binding.

Figure 11. The example with property change support



Because you aren't handling any of the synchronization process, the two private text widget variables are also no longer needed. The `createControls()` method can be modified to use local variables, as shown in Listing 11, thus completing the transformation of the initial example to make full use of JFace data binding.

Listing 11. Changing the code to use local variables

```
GridData gridData =
    new GridData(GridData.FILL_HORIZONTAL);
Text firstText = new Text(shell, SWT.BORDER);
firstText.setLayoutData(gridData);

label = new Label(shell, SWT.NONE);
label.setText("Last:");
Text lastText =
    new Text(shell, SWT.BORDER);
gridData =
    new GridData(GridData.FILL_HORIZONTAL);
lastText.setLayoutData(gridData);

DataBindingContext ctx = createContext();
ctx.bind(firstText,
    new Property(this.person, "first"),
    new BindSpec());
ctx.bind(lastText,
    new Property(this.person, "last"),
    new BindSpec());
```

Section 9. Binding other controls and properties

Text controls aren't the only bindable SWT widgets. All of the standard SWT widgets, such as combos and labels, are available for binding. You can also bind to nonvisual widget properties, such as `enabled`. Copy the code from Listing 12 into the `Person` bean.

Listing 12. Adding `enabled` support to the `Person` bean

```
private
boolean
firstEnabled
= true;

public
boolean
getFirstEnabled()
{
    return
    firstEnabled;
}

public
void
setFirstEnabled(boolean
firstEnabled)
{
    Object
oldVal
=
this.firstEnabled;
```

```
this.firstEnabled
=
firstEnabled;
firePropertyChange("firstEnabled",
\
oldVal,
this.firstEnabled);
}
```

Now modify the `updatePerson()` method in the example.

Listing 13. Modifying the `updatePerson()` method

```
private
void
updatePerson()
{
person.setFirst("James");
person.setLast("Gosling");
person.setFirstEnabled(false);
}
```

Finally, add the bindings shown below to the end of the `createControls()` method.

Listing 14. Binding the labels and enablement

```
ctx.bind(new Property(firstText, "enabled"),
new Property(this.person, "firstEnabled"),
new BindSpec());

ctx.bind(labelFirst,
new Property(this.person, "first"),
new BindSpec());

ctx.bind(labelLast,
new Property(this.person, "last"),
new BindSpec());
```

The new bindings result in the labels of the example changing to the same value as the text widget. The widget for the first field also becomes disabled when you click **Change Name**. Run the example again and test this functionality.

Another interesting effect of these additional bindings can be demonstrated by typing a few characters in the Last field and pressing **Tab**. Notice that the Last label also changes. JFace data binding synchronizes the value of the `Person` bean's last-name field with the widget on focus lost. Because the label is bound to this property, it's also updated.

Section 10. Binding multiple values

So far, you've only bound single values to widgets and widget properties. Many times in an application's UI, more than a single value is needed. The user needs to see a collection of values and select a specific one. This is commonly done with a list or combo box. JFace data binding takes such a requirement into consideration and provides a solution.

To create an example that binds to multiple values, you need a list of multiple values to bind to. This can be accomplished by adding the code in Listing 15 to the `Person` bean you've been enhancing throughout this tutorial. This code creates an `ArrayList` of names, along with a corresponding getter. There is also a convenience method call -- `addName()` -- that takes the `Person` bean's first and last names, concatenates them, and adds them to the list.

Listing 15. Modifications to the Person bean

```
private
List
names;

public
Person(String
first,
String
last) {
this.first
=
first;
this.last
= last;
this.names
= new
ArrayList();
this.names.add("James
Gosling");
this.names.add("Scott
Delap");
this.names.add("Larry
Wall");
}

...

public
List
getAvailableNames()
{
return
this.names;
}

public
void
addName()
```

```

{
  this.names.add(getFirst()
+ " " +
getLast());
  firePropertyChange("availableNames",
  null,
  null);
}

```

Next, modify the code of the `BindingExample` class, as shown in Listing 16. Add the combo- and label-creation code, and the binding code to the `createControls()` method. Then add the button-creation code in the `createButtons()` method.

Listing 16. Modifications to the `BindingExample` class

```

private IObservableValue selectionHolder;
private void createControls(Shell shell) {

    ...

    gridData = new GridData(GridData.FILL_HORIZONTAL);
    gridData.horizontalSpan = 2;
    Label comboSelectionLabel = new Label(shell, SWT.NONE);
    comboSelectionLabel.setLayoutData(gridData);

    gridData = new GridData(GridData.FILL_HORIZONTAL);
    gridData.horizontalSpan = 2;
    Combo combo = new Combo(shell, SWT.BORDER);
    combo.setLayoutData(gridData);

    DataBindingContext ctx = createContext();

    ...

    ctx.bind(
        new Property(combo, SWTProperties.ITEMS),
        new Property(
            this.person,
            "availableNames",
            String.class,
            true),
        new BindSpec());
    this.selectionHolder = new WritableValue(String.class);
    ctx.bind(
        new Property(
            combo,
            SWTProperties.SELECTION),
            selectionHolder,
            new BindSpec());
    ctx.bind(comboSelectionLabel, selectionHolder, new BindSpec());
}

private void createButtons(Shell shell) {

    ...

    gridData = new GridData();
    gridData.horizontalAlignment = SWT.CENTER;
    gridData.horizontalSpan = 2;
    Button button = new Button(shell, SWT.PUSH);
    button.setLayoutData(gridData);
    button.setText("Add Name");
}

```

```
button.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
        person.addName();
        selectionHolder.setValue(
            person.getAvailableNames().get(
                person.getAvailableNames().size() - 1));
    }
});
}
```

The first thing that's different about binding a combo box is that two bindings are needed. This isn't drastically different from binding both the `text` property and the `enabled` property of the text widget in previous examples, except that both bindings are necessary to have a functioning combo box. The code you just pasted in provides the following functionality.

First, a combo control and a label are created. Then the list of available names from the `Person` bean is bound to the combo control. Because this control has properties of both a list of items it contains and a selection, there is no logical default that JFace data binding can pick to create an observable from -- unlike, say, the text widget earlier (which defaults to "text" if no explicit property is specified). As a result, a property has to be explicitly specified when binding a combo. In the case of the first new binding line, `SWTProperties.ITEM` is used to signify that binding the list of available items is desired. When you bind a list, the `Property` object has a couple of extra parameters. The third parameter in its constructor tells the binding context what types of objects are found in the collection. (Part 2 will tell you more about why this is important.) The fourth parameter in the `Property` constructor signifies that this is a collection of values and not a list as a single object.

JFace data binding requires a placeholder for the selection of a combo. This placeholder serves as an external place that references the selection that is not within the control itself. This could be an explicit getter/setter property on a `JavaBean`, but often it needs to be a holder solely for UI purposes, which is the case in this example. As a result, you create a `WritableValue` instance, which implements `IObservableValue` to use as this placeholder. It can then be bound to the combo's `selection` property using the `SWTProperties.SELECTION` constant on the next line. Finally, to show the user that the binding of the selection is working, the same `WritableValue` instance is also bound to a label that changes whenever the selection changes.

The other change to the example is the addition of an **Add Name** button. This button's selection listener invokes the `addName()` method on the `Person` bean, which adds the current name to the list of available names. It then sets the newly added value as the selection value in the selection holder.

Running the example presents a window similar to the one shown in Figure 12. Select a name from the combo box, and the label changes because it's bound to the same selection holder. Next, enter a name in the **First** and **Last** text widgets, and

click **Add Name**. The name is added to the combo, is selected, and appears in the label.

Figure 12. The modified example, now including a combo box



Section 11. The magic behind lists: Observables

In addition to an `IObservableValue` interface, JFace data binding also defines an `IObservableList` interface, as shown in Listing 17. Just as `IObservableValue` creates a generic way to listen to changes of and manipulate values, `IObservableList` specifies a generic way to access a list of objects. There are standard methods to perform evaluations, such as `contains()`, `add()`, `remove()`, `indexOf()`, `iterators()`, etc.

Listing 17. The `IObservableList` interface

```
public
interface
IObservableList
\
extends
List,
IObservableCollection
{

public
void
addChangeListener\
(IListChangeListener
listener);
```

```
public
void
removeChangeListener\
(IListChangeListener\
listener);
public
int
size();
public
boolean
isEmpty();
public
boolean
contains(Object
o);
public
Iterator
iterator();
public
Object[]
toArray();
public
Object[]
toArray(Object
a[]);
public
boolean
add(Object
o);
public
boolean
remove(Object
o);
public
boolean
containsAll(Collection
c);
public
boolean
addAll(Collection
c);
public
boolean
addAll(int
index,
Collection
c);
public
boolean
removeAll(Collection
c);
public
boolean
retainAll(Collection
c);
public
boolean
equals(Object
o);
public
int
hashCode();
public
Object
get(int
index);
public
Object
set(int
```

```

index,
Object
element);
public
Object
remove(int
index);
public
int
indexOf(Object
o);
public
int
lastIndexOf(Object
o);
public
ListIterator
listIterator();
public
ListIterator
listIterator(int
index);
public
List
subList(int
fromIndex,
int
toIndex);
Object
getElementType();
}

```

Building on the `IObservableList` interface is the `JavaBeanObservableList` implementation, of which excerpts are shown in Listing 18. In most cases, with methods like `size()`, it delegates to the list it's adapting. Perhaps the most important method is `updateWrappedList()`. This method takes an old version of the list and a new version, and creates a `Diff` object. This object contains a change log of items that need to be removed and new ones that need to be added. The `Diff` object is then used to synchronize the changes needed to the target implementation of `IObservableList`.

Listing 18. Excerpts from `JavaBeanObservableList`

```

public int size() {
    getterCalled();
    return wrappedList.size();
}

protected void updateWrappedList(List newList) {
    List oldList = wrappedList;
    ListDiff listDiff = Diffs.computeListDiff(
        oldList,
        newList);

    wrappedList = newList;
    fireListChange(listDiff);
}

```

Listing 19 shows an excerpt from the `SWTObservableFactory`. You can see JFace data binding contains the classes `ComboObservableList` and

`ComboObservableValue` for use when generating observables to be used with combo controls. The first adapts the `items` property of a combo to `IObservableList`, and the second adapts the `selection` property to that of the `IObservableValue` interface.

Listing 19. Excerpts from `SWTObservableFactory`

```
if (object instanceof Combo
    && (SWTProperties.TEXT.equals(attribute)
    || SWTProperties.SELECTION.equals(attribute))) {
    return new ComboObservableValue(
        (Combo) object,
        (String) attribute);
} else if (object instanceof Combo
    && SWTProperties.ITEMS.equals(attribute)) {
    return new ComboObservableList((Combo) object);
}
```

Listing 20 shows excerpts from each class. Because a combo can contain either a selection or manually entered text like a text widget when adapted to an `IObservable` value using `ComboObservableValue`, the `get()` method checks this case and calls the appropriate value on the combo control to retrieve the value. Similarly, the `setValue()` method checks which property has been bound and calls the appropriate setter. It then fires a value change event to notify interested parties. `ComboObservableList` delegates to the combo in most instances. Methods like `add()` and `remove()` are the exceptions because a `Diff` of changes must be created for use in the `firstListChange()` method.

Listing 20. Excerpts from `ComboObservableValue` and `ComboObservableList`

```
public class ComboObservableValue extends AbstractObservableValue {
    ...
    public void setValue(final Object value) {
        String oldValue = combo.getText();
        try {
            updating = true;
            if (attribute.equals(SWTProperties.TEXT)) {
                String stringValue =
                    value != null ? value.toString() : ""; //$NON-NLS-1$
                combo.setText(stringValue);
            } else if (attribute.equals(SWTProperties.SELECTION)) {
                String items[] = combo.getItems();
                int index = -1;
                if (items != null && value != null) {
                    for (int i = 0; i < items.length; i++) {
                        if (value.equals(items[i])) {
                            index = i;
                            break;
                        }
                    }
                }
                if (index == -1) {
                    combo.setText((String) value);
                } else {
                    combo.select(index);
                }
            }
        } catch (Exception e) {
            // ...
        }
        update();
    }
}
```

```

        }
    } finally {
        updating = false;
    }

    fireValueChange(Diffs.createValueDiff(oldValue, combo.getText()));
}

public Object doGetValue() {
    if (attribute.equals(SWTProperties.TEXT))
        return combo.getText();

    Assert.isTrue(attribute.equals(SWTProperties.SELECTION),
        "unexpected attribute: " + attribute); //$NON-NLS-1$
    // The problem with a combo, is that it changes the text and
    // fires before it update its selection index
    return combo.getText();
}
}
public class ComboObservableList extends SWTObservableList {
    private final Combo combo;

    ...

    public void add(int index, Object element) {
        int size = doGetSize();
        if (index < 0 || index > size)
            index = size;
        String[] newItems = new String[size + 1];
        System.arraycopy(getItems(), 0, newItems, 0, index);
        newItems[index] = (String) element;
        System.arraycopy(
            getItems(),
            index,
            newItems,
            index + 1,
            size - index);
        setItems(newItems);
        fireListChange(Diffs.createListDiff(Diffs.createListDiffEntry(index,
            true, element)));
    }

    protected int getItemCount() {
        return combo.getItemCount();
    }

    ...
}

```

Section 12. The magic behind lists: ListBinding

The selection observables are kept synchronized using a `ValueBinding` object, which you've seen in detail.

This leaves the lists of values still needing to be synchronized. This duty is

performed by the `ListBinding` class; an excerpt is shown in Listing 21.

This implementation iterates over any differences specified, invoking the appropriate `add()` or `remove()` method on the target `IObservableList` instance.

Listing 21. Excerpts from ListBinding

```
private
IListChangeListener
modelChangeListener
=
new
IListChangeListener()
{
public
void
handleListChange(
IObservableList
source,
ListDiff
diff) {
...

ListDiff
setDiff
=
(ListDiff)
e.diff;
ListDiffEntry[]
differences
=
setDiff.getDifferences();
for
(int i
= 0; i
<
differences.length;
i++) {
ListDiffEntry
entry =
differences[i];
if
(entry.isAddition())
{
targetList.add(
entry.getPosition(),
entry.getElement());
} else
{
targetList.remove(entry.getPosition());
}
}
...
};
```

Section 13. The magic behind lists: Putting it all together

To recap, the `ctx.bind(new Property(combo, SWTProperties.ITEMS` line in Listing 16 tells the binding context to bind the `items` property of a `combo` to the `List` of values returned from calling the `availableNames` getter on the `Person` bean. The context creates implementations of `IObservableList` for each since of the binding relationship. It then creates an instance of `ListBinding` references `IObservableList` to keep them both synchronized when the other changes.

Section 14. Conclusion

This tutorial has introduced a few of the core features in the JFace data binding API. Along the way, you've seen how data binding relieves you from writing the tedious boilerplate synchronization code often necessary in desktop applications. In its place, the JFace data binding API provides a set of interfaces and implementations to generically references JavaBean properties and properties of SWT/JFace widgets.

With this mechanism in place, it can provide synchronization support widgets, such as text controls and labels, as well multivalue lists and combos. You can glue such properties together with a `DataBindingContext.bind()` that supplies the target and model sides of the relationship.

Part 3 of this "[Understanding JFace data binding in Eclipse](#)" series moves on to advanced topics, such as tables, converters, and validation.

Downloads

Description	Name	Size	Download method
Source code	os-ecl-jfacedb2.source.zip	25KB	HTTP

[Information about download methods](#)

Resources

Learn

- Read the ClientJava.com article "[How Many Data Binding Frameworks = A Bad Thing.](#)"
- Learn more about [JFace data binding](#) at the Eclipse Foundation wiki.
- Review the new [JSR 295: Beans Binding](#).
- See Martin Fowler's description of the [Presentation Model](#).
- Expand your Eclipse skills by visiting IBM developerWorks' [Eclipse project resources](#).
- Browse all of the [Eclipse content](#) on developerWorks.
- Visit the developerWorks [Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.
- Stay current with [developerWorks technical events and webcasts](#).

Get products and technologies

- Check out the [JGoodies Binding](#) project.
- And check out the [Spring Rich Client Project \(RCP\)](#).
- Download the [SWTBinding](#).
- Check out the latest [Eclipse technology downloads](#) at IBM [alphaWorks](#).
- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

Discuss

- Get involved in the developerWorks community by participating in [developerWorks blogs](#).

About the author

Scott Delap



Scott Delap is an independent consultant specializing in Java EE and rich Java clients. He has presented papers at JavaOne and is actively involved in the desktop Java community. He is also the administrator of ClientJava.com, a portal focused on desktop Java development. ClientJava.com is frequently featured all over the Web, from JavaBlogs to Sun Microsystems' Web site.