How to build a bean for Servoy
– A step by step tutorial brought to you by Servoy Stuff

PART 4

N. Designing an input bean

In the previous parts of this tutorial, we looked at how to script regular beans, then we built a custom beans out of the JSlider component, and cleaned it up so that it appears just like any other regular Servoy component in the Properties View and the Solution Explorer.

But in part one of this tutorial we said that there were a few possible use of a Slider or Knob component, one being navigation, like we implemented it, another was using it as an input component, and we didn't implement our bean this way. So this time we will look at what it takes to implement both "behaviours" into our bean, depending on a few properties.

So what does it take to use a bean as an "input" bean?
Well, think of the property that is essential to any input component in Servoy and you will have the answer: that's right! We need to set a data provider!

So we will add this variable to our bean:

```
private String dataProviderID;
```

Note that I used dataProviderID as the name of the data provider, - this is a convention used in some of Servoy's interface that we will see later where you will need to implement a method getDataProviderID(); in any case, the Properties view in Servoy will strip the ID from the name of the property, which will appear as "dataProvider", just like any other input component.

Of course we need to add a getter and a setter to our private variable to make it accessible:

```
/**
 * @return the dataProviderID
 */
public String getDataProviderID() {
      return this.dataProviderID;
}

/**
 * @param dataProviderID the dataProviderID to set
 */
public void setDataProviderID(String dataProviderID) {
      this.dataProviderID = dataProviderID;
}
```

Then we will also need easy access to the current record, as opposed to the use we had until now of the current FoundSet only, so let's add a property for that:

```
protected IRecord currentRecord;
```

No need for acessors for this one, but we set it to protected, just in case later we will want to subclass our bean.
We will need to change our updateSlider method (which is called by Servoy by way of the setSelectedRecord() method) to set the value of our slider from a data provider, and we also need to change our stateChanged() method (the one that is called by the JSlider itself when it updates the value as defined by the contract of the ChangeListener interface we implemented) to update our records data provider from the value of the slider.

But first, we will define 2 convenience methods...
The first will simply test that a dataProviderID has been correctly set by the user of our bean:

```
/**
* @return true if a dataProviderID exists and has a value
*/
private boolean hasDataProvider() {
return (getDataProviderID() != null && getDataProviderID().trim().length() > 0);
}
```

The second is a little bit more complicated; in short, it will test that our dataProvider exists in our currentRecord:

```
private boolean isContainedInFoundset(IRecord record, String dataProviderID) {
      if (currentFoundset != null) {

      boolean isContained = false;
      String[] providers = currentFoundset.getDataProviderNames(IFoundSet.COLUMNS);
            if (providers != null) {
                  for (int i = 0; i < providers.length; i++) {
                        if (dataProviderID.equals(providers[i])) {
                              isContained = true;
                              break;
                        }
                  }
            }
            if (isContained && currentFoundset.getSelectedIndex() > -1) {
                  return true;
            }
      }
      return false;
}
```

This method does the following:

- test that the currentFoundset is not null
- get all the dataProvider names from the foundset (the equivalent of the selectedrecord “alldataproviders” property)
- if there are some dataProviders, loop on the array, and check if one of them is equal to our dataProvider (strict equality, - in Java and JavaScript names are usually case sensitive)
- if we have a match and the selected index is on a record, return true
- in any other cases return false.

Armed with these 2 convenience methods, we can now code our updateSlider() method like that:

```
private void updateSlider() {
      if (currentFoundset != null && currentFoundset.getSize() > 0
      && currentRecord != null) {
            if (hasDataProvider()
            && isContainedInFoundset(getDataProviderID())) {
                  Object obj = currentRecord.getValue(getDataProviderID());
                  if (obj != null && obj instanceof Number) {
                        setSliderValue(((Number)obj).intValue());
                  } else {
                        setSliderValue(getMinimumValue());
                  }
            } else {
                  // the behaviour we had before:
                  if (validationEnabled) {
                        setMinimum(1);
                        int max = currentFoundset.getSize();
                        setMaximum(max);
                        setMajorTickSpacing(max-1);
                        int tickSpacing = Math.max(Math.round(max/10),1);
                        setMinorTickSpacing(tickSpacing);
                        setSliderValue(currentFoundset.getSelectedIndex()+1);
                  } else {
                        setMinimum(0);
                        setSliderValue(0);
                  }
            }
      }
}
```

If we have a foundset, not empty, and a current record, we test if we have a data provider and that this data provider is contained in the foundset, this will mean that we can use our bean as an input bean, otherwise we will use our previous behaviour (as a “navigator”).

That’s what our first test does. If the test returns true, then we can retrieve the value of the data provider for the current record as an Object, using the IRecord getValue() method.
Then we can check the value retrieved, and if not null and if the object is indeed a Number (a String won’t do for a slider!), then we call our setSliderValue() with the int value retrieved from our Number object. If the value is null or not a number, then we set it to the minimum value (we need to update our slider in any case).

The rest of the method is the same as the previous version, testing if we are in find mode and updating the value and the display from the currentFoundset.

That’s it for updating our slider when Servoy selects a record!

Now on to our stateChanged() method which is triggered when the user changes the value with the slider:

```
public void stateChanged(ChangeEvent e) {
      if (!ignoreUpdate && currentFoundset != null
      && currentFoundset.getSize() > 0 && currentRecord != null) {
            if (hasDataProvider()
                  && isContainedInFoundset(getDataProviderID())) {

                  if (currentRecord.startEditing()) {
                        currentRecord.setValue(getDataProviderID(), getValue());
                  }

            } else {

                  currentFoundset.setSelectedIndex(getValue()-1);

            }
      }
      ignoreUpdate = false;
}
```

You remember that we only need to take the event into account only if it is not Servoy who is updating the record that triggered it, so that's our first test `if (!ignoreUpdate`
then we test if the foundset is not null and not empty and we have a record to work on:
`&& currentFoundset != null && currentFoundset.getSize() > 0 && currentRecord != null`).

After that, we do the same kind of test sequence than in our updateSlider method: do we have a data provider and is this data provider contained in our foundset? If yes, then we are in "input mode", so we update the data provider's value of the current record with the value of our slider, using the IRecord setValue() method, as easy as that! If not, we are in "navigator" mode and we update the foundset selected index...

Note that as the javadocs of the IRecord tells us, we need to call startEditing() (which return true, we can edit, or false we can't – record lock or any other problem) before any call to setValue() on an IRecord.

Note that we called the IRecord.setValue(String dataProvider, Object value) method, with an int (the result of the getValue() method of the JSlider). This only works because we are using a special feature of Java 1.5 called **auto-boxing**.
Auto-boxing is a great feature that allows us to use primitive types (int, float, double, boolean) where Java is in fact expecting an Object (Integer, Float, Double, Boolean), the compiler will automatically create the relative Object for us, so in our case for example we don't need to do:

```
currentRecord.setValue(getDataProviderID(), new Integer(getValue()));
```

This is only available with java 1.5+, but that's fine since it is our target (Sorry Servoy 3.5.x users, that won't work for you, but the Servoy API interface we use is not compatible anyway!)

Anyway, with our modification, our bean has now been transformed from a simple "Navigator" bean to a "Navigator" and "Input" bean, depending on the data provider String if provided.

Still, we need to add our property to the *ServoySliderBeanInfo* class for it to be accessible by the Servoy Properties View editor, by simply adding this line in the try block of our getPropertyDescriptors() method:

```
liste.add(new PropertyDescriptor("dataProviderID", ServoySlider.class));
```

somewhere before the last two lines of the try block, and the property will be recognized by Servoy.

You could add js_getDataProvider() and js_setDataProvider(String) methods if you wanted to allow access to this property at runtime, but no other “input” component in Servoy has this facility, so we will not add this in our tutorial. Feel free to do it if you want to test it though (and add some lines in the getSample() and getToolTip() while you’re at it).

Now you can deploy your bean and give it a try. If you put it on a form with a data provider on a number database field, and you set the minimum and maximum values to the range you want your users to set the values, it should work already!

O. Making our bean i18n ready

One great thing in Servoy is that it is i18n ready out of the box, all you need is to set up a table for messages keys and values per languages, and with the magical “i18n” prefix the String values will be translated. So it is a good habit whenever you use Strings in your components to provide support for this feature, your international users will thank you!

Now our slider doesn’t use many Strings, but there is one: the “toolTipText” property that we will use as an example on how to do it properly.

When I say that Servoy is i18n ready out of the box, that is true also on the Java side, and in the public API, you will find a class *com.servoy.j2db.Messages* that contains a certain number of static methods that you can use as helpers in your plugins and beans. You can have a look at the javadocs for this class to see what methods there are, but don’t expect much comments in there, apart from disclaimers for a few methods stating “CURRENTLY FOR INTERNAL USE ONLY, DO NOT CALL.”, there is none :{

Anyway this *Messages* class has in particular a useful method which is:

```
public static String getStringIfPrefix(String key)
```

If you pass it a String, it will look for the “i18n:” prefix, and if found will try to retrieve the translated value for the current locale, otherwise it will return the passed String untouched. Meaning you don’t even have to parse the String yourself, if the “i18n:” prefix is there Servoy retrieve the translated value for you (returning the String passed between two exclamation marks if the key was not found).

But to be used correctly we will need to alias our toolTipText property setter and getter.

Why is that?

Because if you set the toolTipText property of the JSlider directly and later wants to retrieve it, it will return the translated value, not the key!!!

For example if you set the toolTipText property in the Property editor to say “i18n:bean.slider.tooltipMessageKey”, it will translate it right away, and you will end up with the translation inside the property editor!

That’s why we need to store the key inside our class, by adding a variable:

```
	private String toolTipMessage;
```

then adding a getter and a setter, like this:

```
	/**
	 * @return the toolTipMessage
	 */
	public String getToolTipMessage() {
		return toolTipMessage;
	}

	/**
	 * @param toolTipMessage the toolTipMessage to set
	 */
	public void setToolTipMessage(String toolTipMessage) {
		this.toolTipMessage = toolTipMessage;
		super.setToolTipText(Messages.getStringIfPrefix(toolTipMessage));
	}
```

This way the getter will always return the same value that was set by the user.
While the setter stores it, and also set the real toolTipText of the superclass (the JSlider) to the translated value (if there is one).

“But wait!”, will you say: our property is called toolTipMessage, not toolTipText, so we need to change our ServoySliderBeanInfo, don’t we? That’s right, grasshopper☺ we do, but you remember how to alias properties in the PropertyDesciptor class, don’t you?

We change the line:

```
	liste.add(new PropertyDescriptor("toolTipText", ServoySlider.class));
```

to these:

```
	pd = new PropertyDescriptor("toolTipMessage", ServoySlider.class);
	pd.setDisplayName("toolTipText");
	liste.add(pd);
```

And voilà! A nice little trick that proved very useful in the end, don’t you think?

Note that it is a design decision on your part whether you want to alter the js_getToolTipText() and js_setToolTipText() to call the new methods, or if you prefer to work with real “translated” message at runtime.

I think the natural way that Servoy’s component does it is to use the “translated” message.
It also makes sense to me, since the scripts are used at runtime they are supposed to deal with runtime properties, not design time properties. But if you prefer to access the real i18n key, you could call getToolTipMessage() and setToolTipMessage() instead of the JSlider direct methods.
Or you can even add a pair of new js_get and js_set methods to access these properties, it’s up to you!
For our tutorial implementation, we will keep calling the JSlider runtime properties.

Now you can deploy and test your bean with i18n toolTipText messages.

P. Designing a useful bean

We could rest on our laurels here, and say that our component is done (for the smart client side anyway), but we could also say that our bean as an input component, well, it's not really that powerful. In fact I tried very hard to find some integers others than IDs in the examples database where I could use the beans as-is in a meaningful way.

I don't know you, but I don't like to feel that I have wasted my time building something not useful at all!

So let's ask ourselves what would be a useful Slider component?

Personally, I would like it to allow input for different kind of values, like floating point numbers for examples, percentages for example now that would be useful! And if I could set the precision as well as the range of values that a user could input with it, I would definitely feel that I have built a useful component.

That's the idea: make the slider capable of working on percentages, or any kind of double value.
You can find a lot more of these numbers in the examples database, so it must be that our bean could be used somehow…

If I look at the "orders" table of the "udm" database that I have based my test form on, I can see one or two nice candidates for a slider: the "amt_discount" or the "pct_discount" values which are percentages, and stored as doubles.

Now it's easier said than done, because the JSlider is made to work on int values and int values only!

I will spare you the details of my searches on how to change that fact but I first thought that it would be possible by changing the model used internally to store the values (min, max, extent and value) from a *DefaultBoundedRangeModel* to a subclass that would have been a *DoubleBoundedRangeModel*, but it didn't work out in the end…
Because the JSlider has lots of internal calls that still relies on int values :{

So what we are going to do now, to make it work on double values, is to trick the JSlider superclass by applying a precision factor n (10^n). Any set value will be multiplied by this precision factor, and any get value will be divided by this precision factor while retaining the n decimals we want.

To do this we will have to alias a few of properties to store the real values, this is true for minimum, maximum, majorTickSpacing, minorTickSpacing and the value itself.
And we will also need the precision factor itself as a property, as well as a boolean to allow the user to decide by just one click whether to use the precision factor or not (to work on decimals or simple integers).

We will see along the way that we will need to alter a few methods but in the end, we will have a useful component, so that's well worth the work!

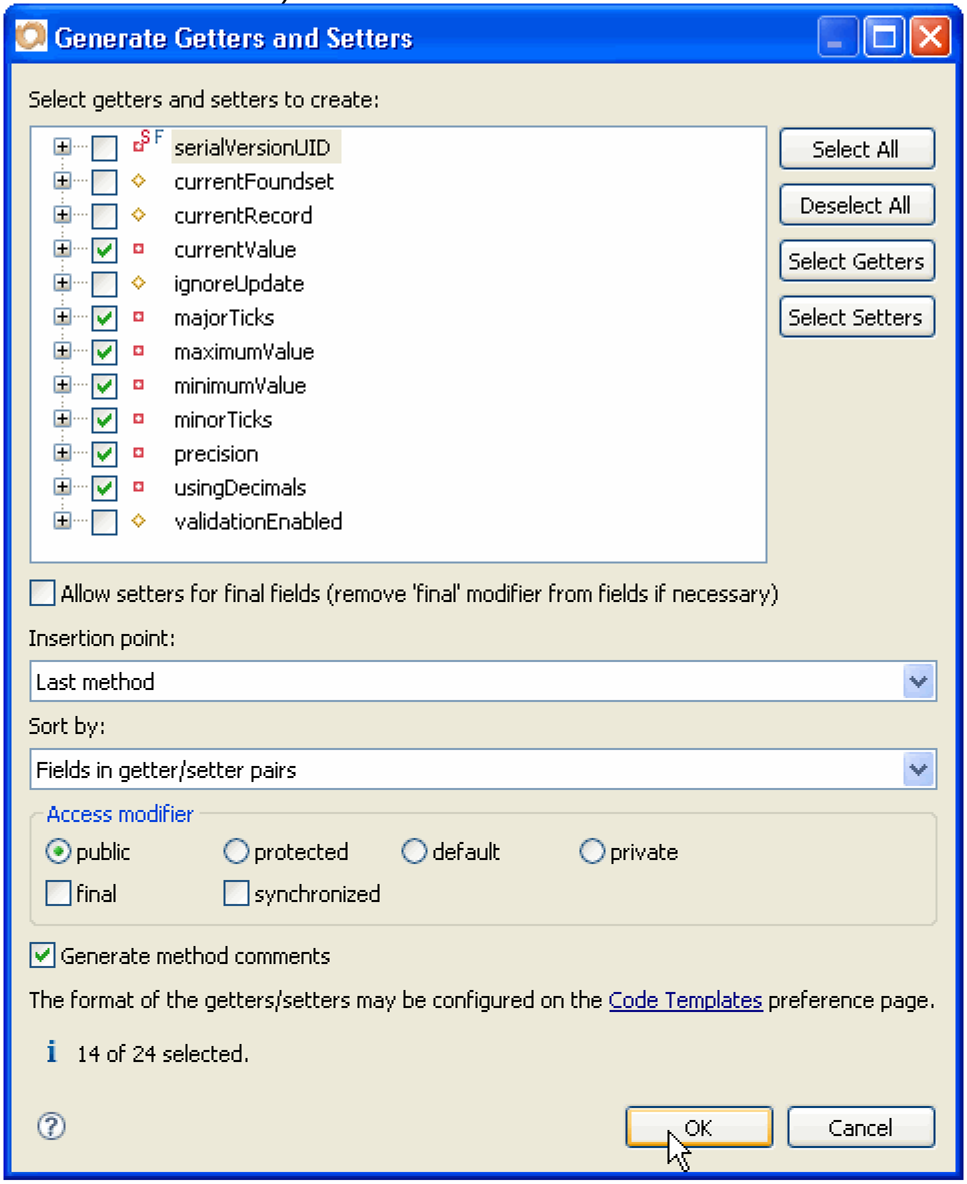
Q. Implementing the useful bean

First, let’s add all our properties, with some default values:

```
private int majorTicks = 50;
private int minorTicks = 10;
private int minimumValue = 0;
private int maximumValue = 100;
private int currentValue = 0;

private int precision = 2;
private boolean usingDecimals;
```

Then let Eclipse generate the getters and setters stubs for all these properties (Source > Generate Getters and Setters...):



Generate Getters and Setters

Select getters and setters to create:

- □ serialVersionUID
- □ currentFoundset
- □ currentRecord
- ☑ currentValue
- □ ignoreUpdate
- ☑ majorTicks
- ☑ maximumValue
- ☑ minimumValue
- ☑ minorTicks
- ☑ precision
- ☑ usingDecimals
- □ validationEnabled

□ Allow setters for final fields (remove 'final' modifier from fields if necessary)

Insertion point: Last method

Sort by: Fields in getter/setter pairs

Access modifier

● public ○ protected ○ default ○ private

□ final □ synchronized

☑ Generate method comments

The format of the getters/setters may be configured on the Code Templates preference page.

14 of 24 selected.

Eclipse will generate 7 getters and 7 setters. And he is a lot faster than us! ☺

Now we need 2 more convenience methods.

The first one will check if our usingDecimals flag is true and if we are using the bean as an input component because otherwise (if used as a "navigator") we certainly don't want to use double values but let the slider work as usual, on integers, so our setter methods will simply pass the values to the superclass (the JSlider).

```
	/**
	 * @return true if is using decimals and a dataProviderID exists
	 */
	public boolean isUsingFactor() {
		return isUsingDecimals() && hasDataProvider();
	}
```

The second utility method is the one creating the precision factor out of the number of decimals that the user will set with the precision property to:

```
	/**
	 * @return the precisionFactor as a double
	 */
	public double getPrecisionFactor() {
		return Math.pow(10.0, getPrecision());
	}
```

Now we can to implement our setters (the getters will stay as generated by Eclipse, they are fine, all we need them to do is return the properties to the Servoy Properties View editor).

Let's start with the setMajorTicks() method:

```
/**
 * @param majorTicks the majorTicks to set
 */
public void setMajorTicks(int majorTicks) {
	if (majorTicks < 0) {
		majorTicks = 0;
	}
	this.majorTicks = majorTicks;
	if (isUsingFactor()) {
		setMajorTickSpacing(majorTicks * (int)Math.round(getPrecisionFactor()));
	} else {
		setMajorTickSpacing(majorTicks);
	}
}
```

As you can see we start by checking that the value is not less than zero (it wouldn't make sense for tick spacing), if it is we set it to 0 (= ignore).

Then we store the new value in our local property.
And then we test if we need to use the factor or not (if usingDecimals is checked, and there is a data provider).
If we need to use the factor, we multiply the value by the precision factor (remember it is a power of ten: 10^n).
If we don't need it, we just forward the call to the JSlider method.

The setMinorTicks() is basically the same (except that it works on minimum, of course ☺):

```
/**
 * @param minorTicks the minorTicks to set
 */
public void setMinorTicks(int minorTicks) {
      if (minorTicks < 0) {
            minorTicks = 0;
      }
      this.minorTicks = minorTicks;
      if (isUsingFactor()) {
            setMinorTickSpacing(minorTicks * (int)Math.round(getPrecisionFactor()));
      } else {
            setMinorTickSpacing(minorTicks);
      }
}
```

Next is the setMinimumValue() method:

```
      /**
       * @param minimumValue the minimumValue to set
       */
      public void setMinimumValue(int minimumValue) {
            if (!hasDataProvider()&& minimumValue < 0) {
                  minimumValue = 0;
            }
            this.minimumValue = minimumValue;
            if (isUsingFactor()) {
                  setMinimum((int)(minimumValue*getPrecisionFactor()));
            } else {
                  setMinimum(minimumValue);
            }
      }
```

The only difference for minimum is that we can use values less than zero (and even for maximum if we want to work with a whole range of values less than zero), except that if there is no data provider (meaning we use the bean as a “navigator”, remember?) it wouldn’t make sense, so that’s why the test is a little bit different for this one. The rest of it is based on the same principle.

And our setMaximumValue() method looks just the same:

```
      /**
       * @param maximumValue the maximumValue to set
       */
      public void setMaximumValue(int maximumValue) {
            if (!hasDataProvider()&& maximumValue < 1) {
                  maximumValue = 1;
            }
            this.maximumValue = maximumValue;
            if (isUsingFactor()) {
                  setMaximum((int)(maximumValue*getPrecisionFactor()));
            } else {
                  setMaximum(maximumValue);
            }
      }
```

And the setCurrentValue() method will look familiar to you too:

```
	/**
	 * @param currentValue the currentValue to set
	 */
	public void setCurrentValue(int currentValue) {
		if (!hasDataProvider()&& currentValue < 0) {
			currentValue = 0;
		}
		this.currentValue = currentValue;
		if (isUsingFactor()) {
			setdoubleValue((int)( currentValue *getPrecisionFactor()));
		} else {
			setValue(currentValue);
		}
	}
```

Now the setPrecision() is a little bit more interesting:

```
	/**
	 * @param precision the precision to set
	 */
	public void setPrecision(int precision) {
		int currentPrecision = getPrecision();
		if (precision < 0) {
			precision = 0; // can't be negative!
		}
		this.precision = precision;
		if (currentPrecision != precision) {
			updateModel();
		}
	}
```

Here, we first store the previous precision value into a temporary variable.

Then we test if the precision is less than 0 (which wouldn't make sense because when rounded to an integer any x * 10^{-n} would always be 0). A value of 0 is admissible though, since our precision factor in that case would be 10^{0} = 1, meaning no change to our integers.
After we tested for admissible values, we set our property to the new value.
And then we test if the precision has changed from the previous precision. If it has, we need to recalculate all the other values...

That's why I made a call to a method that I called updateModel(), it doesn't exist yet, but that's a good use of the quick fix powerful trick: let Eclipse create it for us, and we can implement the stub like that:

```
	/**
	 * Reset the values multiplied by the precision factor if needed
	 */
	protected void updateModel() {
		setMinimumValue(getMinimumValue());
		setMaximumValue(getMaximumValue());
		setMajorTicks(getMajorTicks());
		setMinorTicks(getMinorTicks());
		setCurrentValue(getCurrentValue());
	}
```

All this method does is to retrigger the calculation of the slider values by calling them with the properties we stored, so if we had a factor of 10^2 before and we now have a factor of 10^3 you can easily deduce that since the factor has changed in the meantime, values will be recalculated.

For example, if for maximumValue we had 100, the real maximum of the JSlider was in fact 100*100, but calling setMaximumValue() again with our stored (100) will now set it to 100*1000.
I told you I was full of tricks ;-)

And our last setter method will also make use of our new updateModel():

```java
/**
 * @param usingDecimals the usingDecimals to set
 */
public void setUsingDecimals(boolean newUsage) {
      boolean usingDecimalsBefore = isUsingDecimals();
      this.usingDecimals = newUsage;
      if (usingDecimalsBefore != newUsage) {
            updateModel();
      }
}
```

Just like in the setPrecision() method, it is important here that we retrigger all the calculations if this flag changed.

OK, we have all our setters in place; let's see how we are going to use the precision factor to retrieve a decimal value.

Now might be a good time to let you know (if you don't already) about what I would call the "**uncertainty principle**" of floats and doubles in Java.
This is a well known issue that has been discussed for years in Java circles (and in computer circles long before Java was even born, actually). The thing is that there exist some numbers that can be written with finite-length in decimal (base 10), but will repeat infinitely in binary. When that happens, the numbers are rounded to the nearest exactly representable binary value.
See the following thread post for some funny examples of this principle:
http://www.velocityreviews.com/forums/t139008-java-double-precision.html

Enough of the theory, just keep in mind that any decimal number in most computer languages is often different internally from what it seems. One of the options to deal with that in Java is the use of the *BigDecimal* class, another one (maybe not as optimal but a lot easier) is the one I will use here: format a String with the exact precision we want from a float or a double, then parse it again to get the right number.

So this is the idea behind this convenience method that we will use to set the value of the data provider or to return the value to JavaScript with the exact number of decimals:

```java
/**
* returns the value with correct precision
*/
public Number getNumberValue() {
      NumberFormat formatter = NumberFormat.getNumberInstance();
      formatter.setMaximumFractionDigits(getPrecision());
      String s = formatter.format(getValue()/getPrecisionFactor());
      try {
            return formatter.parse(s);
      } catch (ParseException ex) {}
```

```
        return null;
    }
```

I use a NumberFormat, set its precision to our precision value, then format our value divided by the precision factor (we want to retrieve the decimal value, not the integer value that is used internally by the slider), then once I have a String correctly formatted, I parse it back into a number, and this time for sure we will have the right number of decimals.

If you don’t believe me, try the bean without this method (simply returning getValue()/getPrecisionFactor()) and you will see how this is useful…

OK, we’ve got our little utility method that will ensure the right number of decimals when we want to retrieve the value so let’s use it to set the value of our data provider…
It all takes place in the stateChanged() method of course:

```
public void stateChanged(ChangeEvent e) {
    if (!ignoreUpdate && currentFoundset != null
    && currentFoundset.getSize() > 0 && currentRecord != null) {
        if (hasDataProvider()
            && isContainedInFoundset(getDataProviderID())) {
            if (isUsingFactor()) {

            currentRecord.setValue(getDataProviderID(), getNumberValue());

            } else {
                currentRecord.setValue(getDataProviderID(), getValue());
            }
        } else {
            currentFoundset.setSelectedIndex(getValue()-1);
        }
    }
    ignoreUpdate = false;
}
```

The only thing that changed here is the way we first test if we are using a precision factor, and if so who we set our data provider value to the number value from our convenience method.

Another thing needs to be changed: when Servoy updates the selected record, we need to retrieve the value, not as an integer, but as a double, and multiply it by the precision factor. Before we do that, we need a way to set the value from a double and not from an integer.
Looking at the implementation of the setCurrentValue(), which uses an int as parameter, we see that it will not do, except that there is a part of the inner code of that method that will be useful for that purpose:

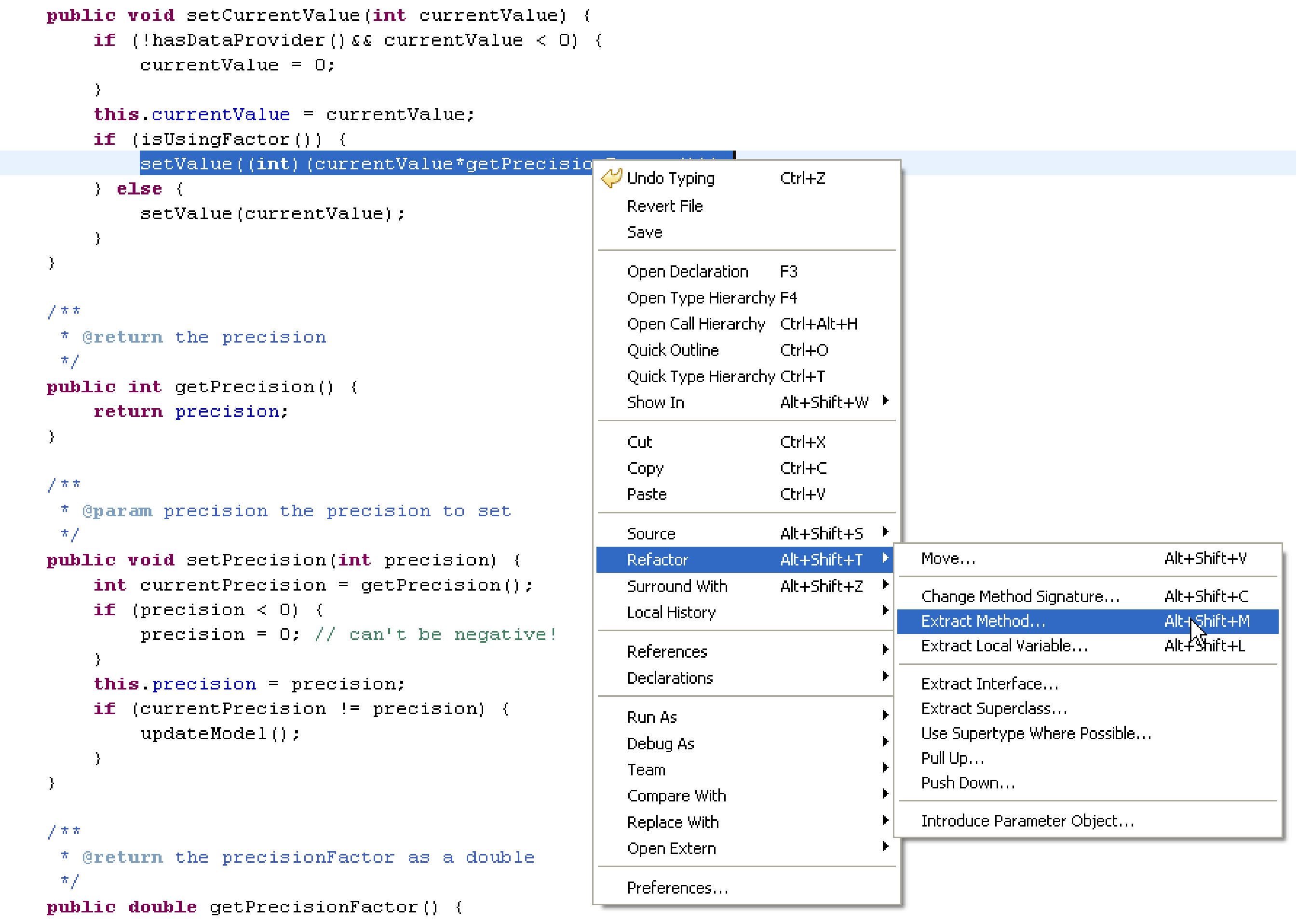
```
    setValue((int)(currentValue*getPrecisionFactor()));
```

It does what we need, and the currentValue could be a double instead of an int.
This is where I can start introducing a very powerful way of programming (if you don’t know it already) which is called **refactoring**. Now you probably have heard about it, you know that it is just a fancy word to say “reorganize the code” to make it more efficient, more logical, more maintainable (the DRY principle again!) and you have probably used it numerous times, but being in Eclipse you should know that there are lots of powerful tools built-in that will help you do it in a very efficient and effortless way.

For example, right here, I just saw a piece of code that I would like to reuse from another point in my class (or even another class maybe). The very bad reflex would be to copy/paste it somewhere else, and bingo! Except that later, in a month, in a year, in a decade (!), when you will have to modify that famous piece of code, you will have to perform searches and replaces on maybe thousands of java files, and that's not part of the fun of Java programming IMHO!

But knowing Eclipse, you will do much better than copy/paste: you will ask Eclipse to refactor your code and place it in a nice new method to be used more conveniently.

It's as easy as, selecting the piece of code you want to reuse, do a right click (or go to the menu) and choose "Refactor > Extract Method..."



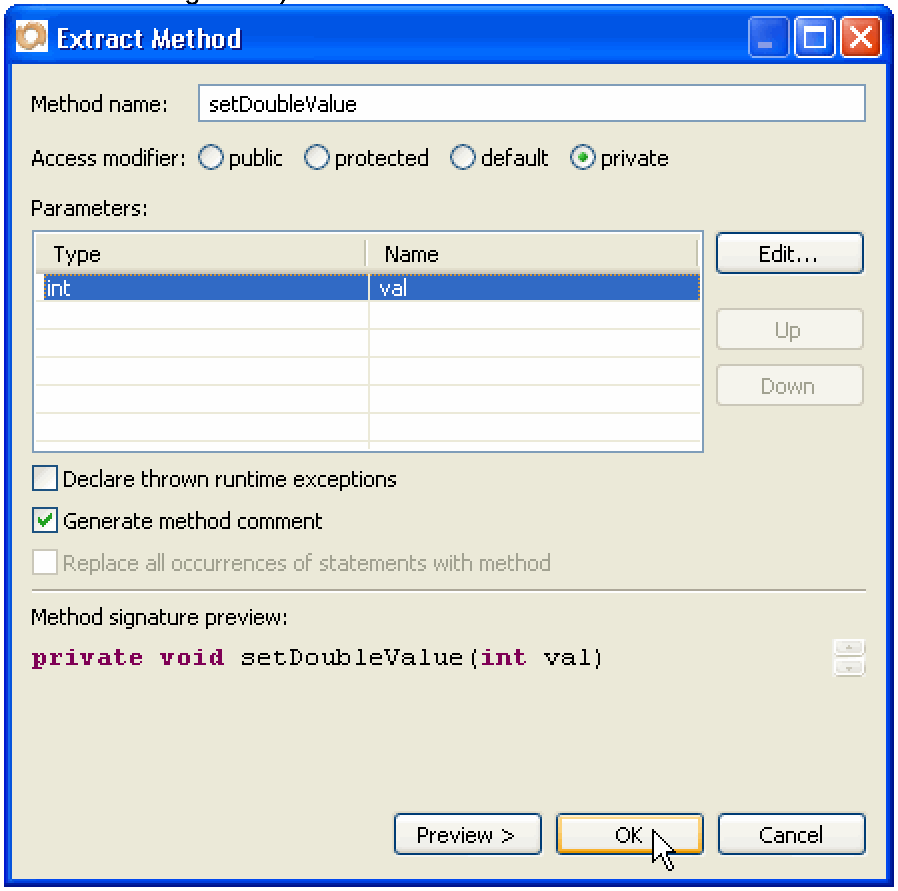
```
public void setCurrentValue(int currentValue) {
    if (!hasDataProvider() && currentValue < 0) {
        currentValue = 0;
    }
    this.currentValue = currentValue;
    if (isUsingFactor()) {
        setValue((int)(currentValue*getPrecisio
    } else {
        setValue(currentValue);
    }
}

/**
 * @return the precision
 */
public int getPrecision() {
    return precision;
}

/**
 * @param precision the precision to set
 */
public void setPrecision(int precision) {
    int currentPrecision = getPrecision();
    if (precision < 0) {
        precision = 0; // can't be negative!
    }
    this.precision = precision;
    if (currentPrecision != precision) {
        updateModel();
    }
}

/**
 * @return the precisionFactor as a double
 */
public double getPrecisionFactor() {
```

You will be asked to give your new method a name, you can choose to make it public, protected, default (package protected) or private, and you can edit the name of the parameter (I chose “val” to make it more generic):



Method name: setDoubleValue

Access modifier: ○ public ○ protected ○ default ● private

Parameters:

Type	Name
int	val

☐ Declare thrown runtime exceptions

☑ Generate method comment

☐ Replace all occurrences of statements with method

Method signature preview:

```
private void setDoubleValue(int val)
```

Once you click “OK”, you will see that Eclipse has generated the method for you, and that it also has changed the extracted code (previously selected) by a call to the new method! In fact this “Extract Method…” is so powerful that it is also capable of looking everywhere in your class to see if there are no other similar pieces of code and if it finds some it will propose to replace them by a call to the new method! Now that’s refactoring the way I like it: no searches and replaces, almost everything is done for you (told you I was lazy ;-)

So the result of our little refactoring example is:

```
	/**
	 * @param val
	 */
	private void setDoubleValue(int val) {
		setValue((int)(val*getPrecisionFactor()));
	}
```

We will modify it slightly to accept double instead, since that was the original idea, so it will now be:

```
	/**
	 * Set the slider’s value from a double using the precision factor
	 * @param val double
	 */
	private void setDoubleValue(double val) {
		setValue((int)(val*getPrecisionFactor()));
	}
```

That’s it, and if you look at the setCurrentValue() method you can see that it is now changed to:

```
    public void setCurrentValue(int currentValue) {
        if (!hasDataProvider()&& currentValue < 0) {
            currentValue = 0;
        }
        this.currentValue = currentValue;
        if (isUsingFactor()) {
            setDoubleValue(currentValue);
        } else {
            setValue(currentValue);
        }
    }
```

You can use and abuse of this facility as well as the other refactoring methods (Extract interface, Change method signature, Rename) all these are really powerful tools that, once you are used to, you will really miss when not in the Java editor (for example in the Servoy JavaScript editor or any straight text editor). Most of these features works for an entire project and will automatically change the code in many java files in one go, that’s power straight under your mouse!

Of course, “with power comes responsibility” (quoted from Spiderman, I know my classics☺), so use it as much as you need, but know what you doing: your code, if used by external programs, could break. Especially when you change public method names for example (a case where @deprecated annotations and comments will be useful)…

Anyway, back to our implementation, we wanted to use this piece of code to set the value when Servoy is updating our selected record, so back to our setSelectedRecord() method and we’ll replace this line:

```
    setSliderValue(((Number)obj).intValue());
```

By this one:

```
    setSliderValue((Number)obj);
```

This will give us more options by passing the Number directly as a parameter (a Number can be anything: an int/Integer, a float/Float or a double/Double, a Big Decimal if you decide to use it…)

But our setSliderValue will now have to be:

```
    private void setSliderValue(Number x) {
        ignoreUpdate = true;
        if (isUsingFactor()) {
            setDoubleValue(x.doubleValue());
        } else {
            setValue(x.intValue());
        }
    }
```

There’s our call to the setDoubleValue() method that we just produced by refactoring.

Note also that thanks to auto-boxing our setSliderValue can still be called with an int, like in the call to:

```
    setSliderValue(0);
```

OK, we are pretty close now. Our slider class can use any kind of value with a precision factor.

We can add some “js_” methods to the precision and usingDecimals properties for JavaScript access:

```
      public void js_setPrecisionFactor(int p) {
            setPrecision(p);
      }
      public int js_getPrecisionFactor() {
            return getPrecision();
      }
      public void js_setUsingDecimals(boolean newUsage) {
            setUsingDecimals(newUsage);
      }
      public boolean js_isUsingDecimals() {
            return isUsingDecimals();
      }
```

And we don’t forget to add the relevant lines to the getToolTip() method:

```
            } else if ("precisionFactor".equals(methodName)) {
                  return "Set / Get the precision factor (number of decimals)";
            } else if ("usingDecimals".equals(methodName)) {
                  return "Set / get the usingDecimals flag";
```

And to the getSample() method:

```
      } else if ("precisionFactor".equals(methodName)) {
            buff.append("\tvar percentFactor = %%elementName%%.precisionFactor;\n");
            buff.append("\t%%elementName%%.percentFactor = 3; (3 decimals)");
      } else if ("usingDecimals ".equals(methodName)) {
            buff.append("\tvar useDecimals = %%elementName%%.usingDecimals;\n");
            buff.append("\t%%elementName%%.usingDecimals = !useDecimals;");
```

This will let our users know a bit more about these new properties.

We also need to change a our js_getValue() and js_setValue() methods to take the precision factor into account:

```
public void js_setValue(Number val) {
      if (isUsingFactor()) {
            setDoubleValue((val == null) ? getMinimumValue() : val.doubleValue());
      } else {
            setValue(val.intValue());
      }
}
public Number js_getValue() {
      if (isUsingFactor()) {
            return getNumberValue();
      } else {
            return getValue();
      }
}
```

Note that I checked for null in the js_setValue() method - when calling a method on an object coming from the outside, that’s always safer!
Note also the use of our setDoubleValue() method again in the js_setValue() method – isn’t it nice when everything is falling into place?
Note finally the use of our convenience getNumberValue() – what did I just say?

Of course we will also need to change the js_get and js_set of the other “aliased” values:

```
        public void js_setMajorTickSpacing(int majorTickSpacing) {
              setMajorTicks(majorTickSpacing);
        }
        public int js_getMajorTickSpacing() {
              return getMajorTicks();
        }
        public void js_setMaximum(int maximum) {
              setMaximumValue(maximum);
        }
        public int js_getMaximum() {
              return getMaximumValue();
        }
        public void js_setMinimum(int minimum) {
              setMinimumValue(minimum);
        }
        public int js_getMinimum() {
              return getMinimumValue();
        }
        public void js_setMinorTickSpacing(int minorTickSpacing) {
              setMinorTicks(minorTickSpacing);
        }
        public int js_getMinorTickSpacing() {
              return getMinorTicks();
        }
```

That’s quite a lot of code we changed all in all.

And still a last thing is to update is our *ServoySliderBeanInfo* class with the new properties.

So we add these lines in the try block:

```
        liste.add(new PropertyDescriptor("usingDecimals", ServoySlider.class));
        pd = new PropertyDescriptor("precision", ServoySlider.class);
        pd.setDisplayName("precisionFactor");
        liste.add(pd);
```

And we alter all the normal properties of the JSlider and replace them with our own, while putting some aliases in place (so that everything will look the same to our users):

```
        pd = new PropertyDescriptor("majorTicks", ServoySlider.class);
        pd.setDisplayName("majorTickSpacing");
        liste.add(pd);
        pd = new PropertyDescriptor("maximumValue", ServoySlider.class);
        pd.setDisplayName("maximum");
        liste.add(pd);
        pd = new PropertyDescriptor("minimumValue", ServoySlider.class);
        pd.setDisplayName("minimum");
        liste.add(pd);
        pd = new PropertyDescriptor("minorTicks", ServoySlider.class);
        pd.setDisplayName("minorTickSpacing");
        liste.add(pd);
        pd = new PropertyDescriptor("currentValue", ServoySlider.class);
        pd.setDisplayName("value");
        liste.add(pd);
```

It’s time to deploy and have fun with it!

R. Mama, what happened to my labels?

If you use the bean in Servoy it should work, try using:

- dataProvider=amt_amount (based on the “orders” table of the “udm” database)
- minimum=0
- maximum=100
- majorTickSpacing=50
- minorTickSpacing=10
- value=0
- precisionFactor=2
- usingDecimals=true
- paintTicks=true

Try changing some of these values, try negative values, try with or without checking the usingDecimals flag, try using snapToTicks, try with or without a dataProvider, and see what happens.
Everything should be ok – if not, check with the provided code in the downloadable Eclipse project.

Now try with the previous values and set paintLabels=true.

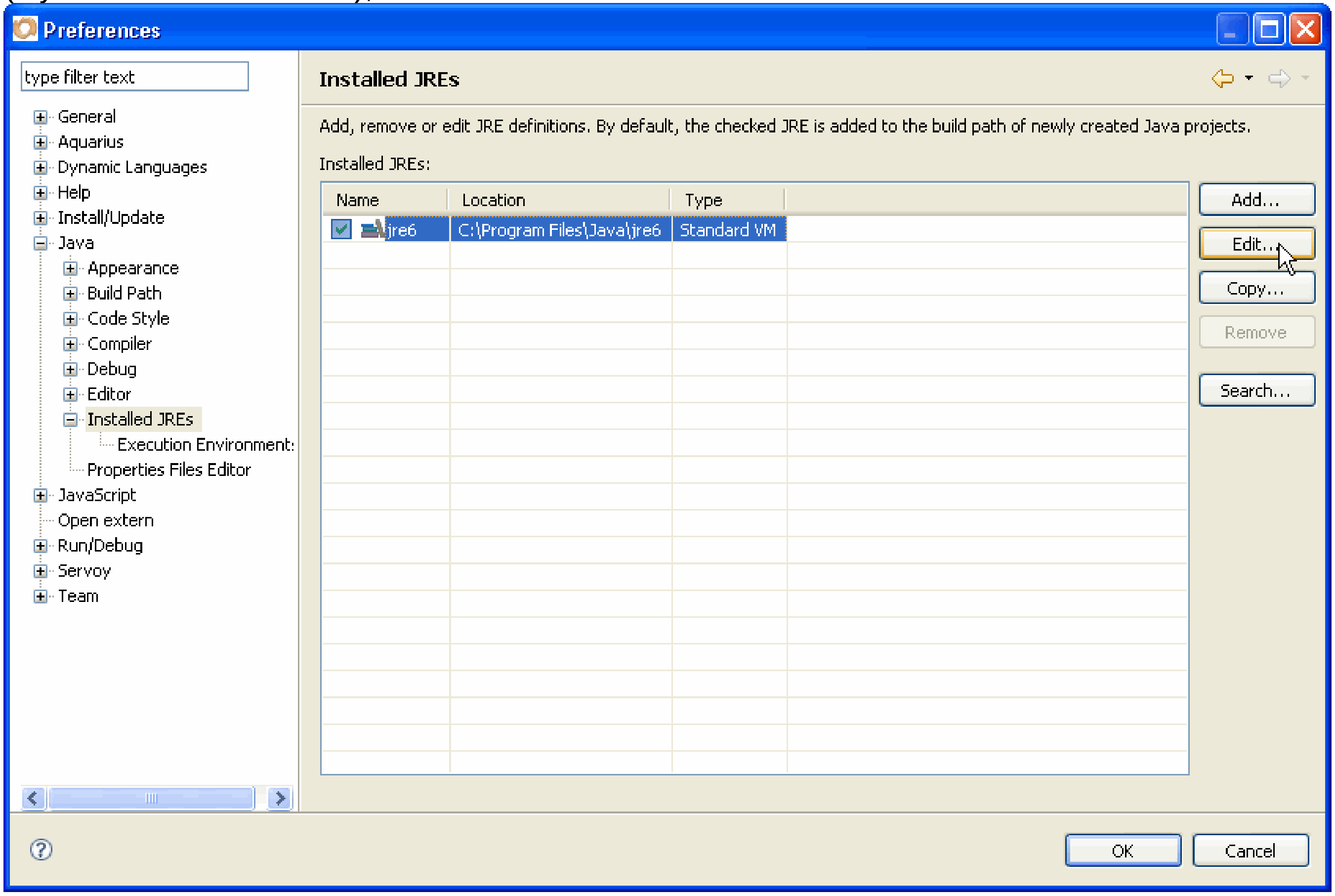
Wow! Mama, what happened to my labels?
Well, the labels by default are drawn on the major ticks and they use the value of these major ticks, but remember that with a precisionFactor of 2 we are in fact multiplying everything by 100, so we our major tick’s values are set at 0 – 5000 – 10000!

Now even if this is perfectly normal, it sure will confuse our users!
We definitely need to fix this.

There are different ways to fix this problem, and we will first have a look at the sources of the JSlider to see how the labels are implemented to get a better understanding.

Note: To be able to do that and to browse in any of the Java sources of Swing, all you need to do is to declare the location of the sources of the JRE in Servoy.
Of course you need to have them on your hard disk already, so if you don’t, go to the Sun website (while it is still Sun and not Oracle ;-) and download them, they are freely available.

Now that you have the sources as a zip file, in Servoy/Eclipse go to the preferences dialog/panel (menu Window > Preferences... > Java > Installed JREs) and check the currently installed JRE of your choice (if you have more than one), and click edit:



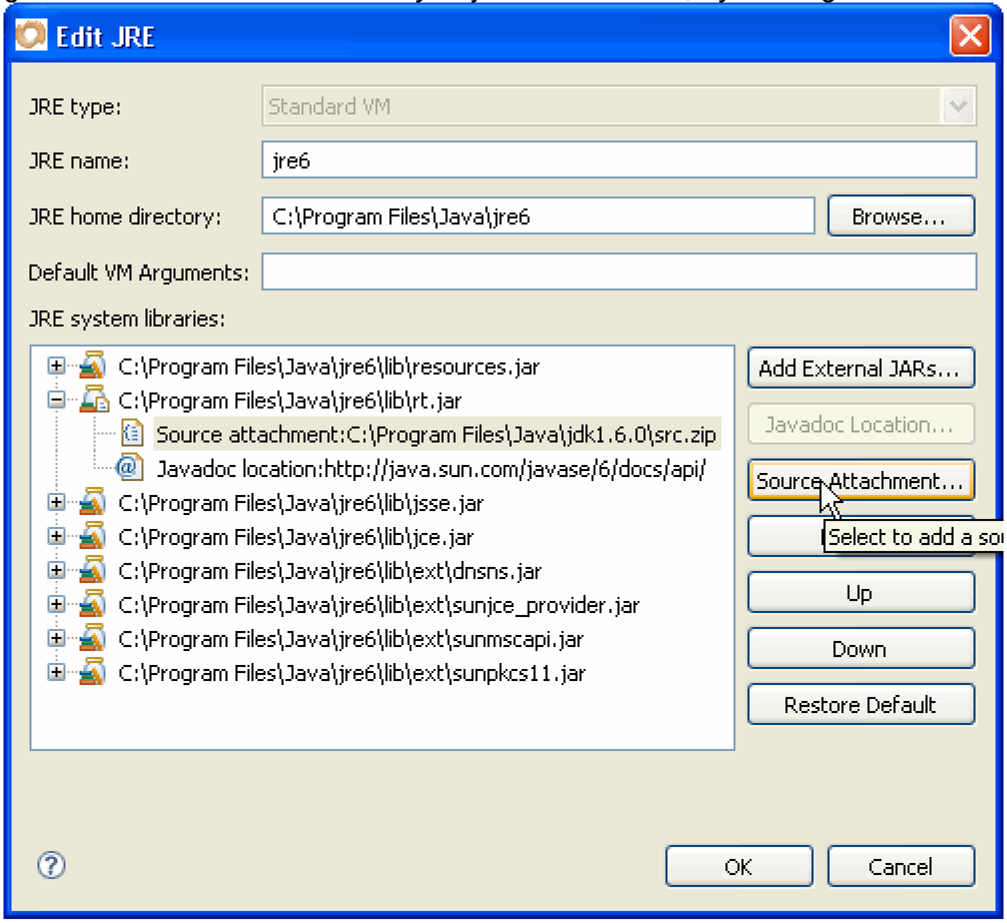
Installed JREs

Add, remove or edit JRE definitions. By default, the checked JRE is added to the build path of newly created Java projects.

Installed JREs:

Name	Location	Type
☑ jre6	C:\Program Files\Java\jre6	Standard VM

In the “Edit JRE” dialog, open the “rt.jar” lib node click on the line called “Source attachment:” and go get the source attachment that you just downloaded, by clicking on the “Source Attachment…” button:



JRE type: Standard VM
JRE name: jre6
JRE home directory: C:\Program Files\Java\jre6
Default VM Arguments:
JRE system libraries:

```
C:\Program Files\Java\jre6\lib\resources.jar
C:\Program Files\Java\jre6\lib\rt.jar
    Source attachment:C:\Program Files\Java\jdk1.6.0\src.zip
    Javadoc location:http://java.sun.com/javase/6/docs/api/
C:\Program Files\Java\jre6\lib\jsse.jar
C:\Program Files\Java\jre6\lib\jce.jar
C:\Program Files\Java\jre6\lib\ext\dnsns.jar
C:\Program Files\Java\jre6\lib\ext\sunjce_provider.jar
C:\Program Files\Java\jre6\lib\ext\sunmscapi.jar
C:\Program Files\Java\jre6\lib\ext\sunpkcs11.jar
```

From now on, you will be able to browse the source of the JRE just like any of your own sources! For example, if you go at the top of your *ServoySlider* class where you have the declaration:

```
public class ServoySlider extends JSlider implements IServoyAwareBean,
ChangeListener, IScriptObject {
```

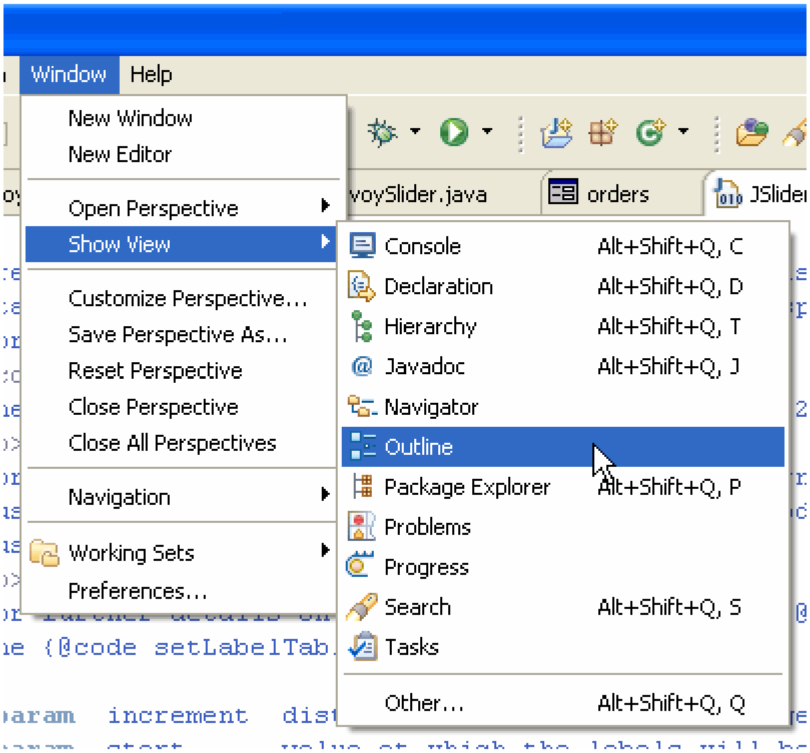
If you put your mouse over the “JSlider” word while holding down the CTRL key (CMD for Mac aficionados), you will see that the word gets underlined like a link. Now click on this and you will browse straight into the source of the JSlider Swing class!

Once in the JSlider java source, you can see that it’s a bit crowded, but don’t worry: there are more comments than code, really!

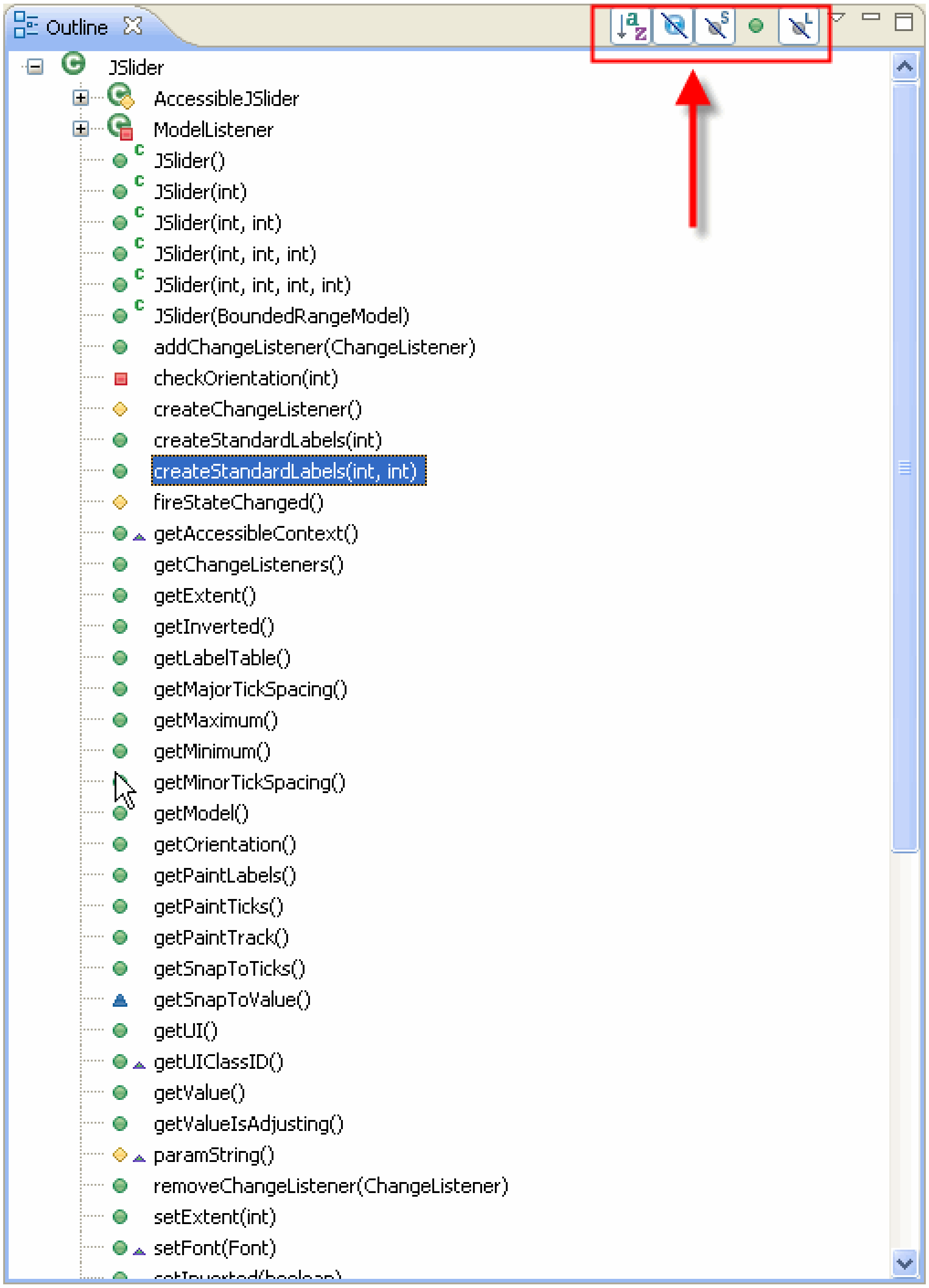
It is the kind of comments that are used to generate the javadocs, using the “javadocs” tool.

If only Servoy had made the kind of effort that you can see here in this class (or any of the JRE classes) for the code of the public API, I will have (almost☺) no business doing these silly tutorials ;-)

Now what can you do to have a quick overlook at a class when you want to go straight to some method?
One way is to use the Outline view. – If this view is not open in your perspective, it is not too late, open it via to the menu “Window > Show View > Outline”:



If it is not already there you can drag it to the side, and you will be able to see the outlined content of the class: imports, properties, methods… Depending on what is filtered - you can filter what should appear bt using the buttons on top of the view, try them, they won't bite!



- JSlider
  - AccessibleJSlider
  - ModelListener
  - JSlider()
  - JSlider(int)
  - JSlider(int, int)
  - JSlider(int, int, int)
  - JSlider(int, int, int, int)
  - JSlider(BoundedRangeModel)
  - addChangeListener(ChangeListener)
  - checkOrientation(int)
  - createChangeListener()
  - createStandardLabels(int)
  - createStandardLabels(int, int)
  - fireStateChanged()
  - getAccessibleContext()
  - getChangeListeners()
  - getExtent()
  - getInverted()
  - getLabelTable()
  - getMajorTickSpacing()
  - getMaximum()
  - getMinimum()
  - getMinorTickSpacing()
  - getModel()
  - getOrientation()
  - getPaintLabels()
  - getPaintTicks()
  - getPaintTrack()
  - getSnapToTicks()
  - getSnapToValue()
  - getUI()
  - getUIClassID()
  - getValue()
  - getValueIsAdjusting()
  - paramString()
  - removeChangeListener(ChangeListener)
  - setExtent(int)
  - setFont(Font)
  - setInverted(boolean)

Now, in this view we can quickly see that there is a method (there are 2 even) called "createStandardLabels()". Click on it, and the Java Editor should show you its code now.

As you can see there is a fair amount of code in there, there is also a protected inner class…
Of course this code is not modifiable, it's the JRE source, so I wouldn't mess with it anyway ☺

Does this mean that we will have to copy/paste all of this code into our ServoySlider class to modify its behaviour? Nah! And stop thinking that copy/paste is a good idea! It is always the worst you can have!

No, we are here to see how the labels are created, that's all! It will help us create our own implementation based on our understanding. In fact what is interesting is to see that the labels are created from a Hashtable of Integer keys / JLabel subclass values, look at the following method located almost at the end of our createStandardLabels() method, the last method inside the SmartHashTable inner class:

```
void createLabels() {
for ( int labelIndex = start; labelIndex<=getMaximum(); labelIndex += increment ) {
   put(new Integer(labelIndex),new LabelUIResource(""+labelIndex,JLabel.CENTER ) );
}
}
```

See how they iterate from start (which is the minimum value), to maximum, by increment (the majorTickSpacing), to put into the "SmartHashTable" itself the key-value pairs of Integer(index) with a LabelUIResource(index, centered), where LabelUIResource is nothing but a subclass of JLabel.

We now know how they did it, so you can guess that we will need to build our own table (of rather fill the already existing table) with our keys based on the index, and a label of based on the index divided by our precisionFactor.

So here is the code we will add to our class:

```
/**
 * Updates the label table if needed
 */
private void updateLabels() {
      Hashtable<Integer, JLabel> table = (Hashtable)getLabelTable();
      if (table != null) {
            table.clear();
            for ( int labelIndex = getMinimum();
                  labelIndex <= getMaximum();
                        labelIndex += getMajorTickSpacing() ) {
                  int val = (isUsingFactor())
                        ? (int)Math.round(labelIndex/getPrecisionFactor())
                        : labelIndex;
            table.put( new Integer(labelIndex), new JLabel(""+val, JLabel.CENTER ));
            }
            setLabelTable(table);
      }
}
```

That's very much inspired by the JRE method, don't you think?
The difference is that I check that the table has already been created, if so I clear it first, and then I add my own custom values to it. If the slider uses the precisionFactor, then I set the label to the index/precisionFactor, and in the end I set JSlider's labelTable to my customized table.

OK, fine, but we will need to call this every time one of the key elements in the loop changes, basically, whenever the minimum, maximum or majorTickSpacing properties changed. And of course we also need to call it if the usingDecimals or precision properties changed.

We already have a method that is called whenever usingDecimals or precision changes, and that's our updateModel() method, so we only add a call to our updateLabels at the end of this method:

```
      protected void updateModel() {
            setMinimumValue(getMinimumValue());
            setMaximumValue(getMaximumValue());
            setMajorTicks(getMajorTicks());
            setMinorTicks(getMinorTicks());
            setCurrentValue(getCurrentValue());
            updateLabels();
      }
```

But for the minimum, maximum and majorTickSpacing let's do it differently, so that you can see how it's usually done in the JRE: in the JSlider sources, if you look at the setMaximum, setMinimum methods (and basically all the setters for the main properties of the class), you will see that there is generally a call to a firePropertyChange() method, for example for setMaximum:

```
firePropertyChange( "maximum", new Integer( oldMax ), new Integer( maximum ) );
```

And of course the firePropertyChange() is the method that (guess what?) fires a propertyChangeEvent, with the name of the property, its old value and its new value.

So we are going to listen to these events, by implementing the PropertyChangeListener interface in our ServoySlider, so our final (for now!) class signature will be:

```
public class ServoySlider extends JSlider implements IServoyAwareBean,
ChangeListener, IScriptObject, PropertyChangeListener {
```

Use the usual tricks when you add an interface (CTRL + space / CMD + space) to add the import and then quick fix to tell Eclipse to create the stub methods of the interface.

We are lucky, there is only one method in this interface, and we are going to implement it as follows:

```
public void propertyChange(PropertyChangeEvent evt) {
      if (isUsingFactor() && getPaintLabels() && getMajorTicks() > 0) {
            String propertyName = evt.getPropertyName();
            if ("minimum".equals(propertyName)
                        || "maximum".equals(propertyName)
                              || "majorTickSpacing".equals(propertyName)) {
                  updateLabels();
            }
      }
}
```

We check that we are using a precision factor, if we need to paint the labels and if the major ticks have been set to a value. If so, we will update the labels only if the properties changed are one of the 3 we use in our label creation loop: minimum, maximum and majorTickSpacing. So we check that the propertyName of the propertyChangeEvent is one of these, before we call our updateLabels().

Believe it or not, that concludes our implementation of a useful slider bean for the smart client, usable as a navigator but also as an input bean with a wide range of possible precision values.

You can now deploy your jar, and test it in Servoy, everything should be ok, including the labels!

You can possibly enhance this little thing, and I will leave that to you. If you make something nice out of it, please release it as Open Source and give back to the community, if you want I will host your enhancements with full acknowledgement of your authorship.

As usual, you will find the complete Eclipse project on the Servoy Stuff web site, here:
http://www.servoy-stuff.net/tutorials/utils/t02/v3/ServoySlider_EclipseProject.zip
(Get rid of the previous project of the same name and import in Eclipse)

The compiled bean (targeted for java 1.5) will be available here:
http://www.servoy-stuff.net/tutorials/utils/t02/v3/servoy_slider.jar
(Put in you /beans folder)

And the little “beans_tests” solution updated to use the new bean in situation will be available at:
http://www.servoy-stuff.net/tutorials/utils/t02/v3/beans_tests-v3.zip
(Unzip and import in Servoy 4.1.x)

Hope you enjoyed this tutorial and that you learned a few useful things from it.

Feel free to send comments, suggestions, questions, etc… to me on the Servoy Stuff site.

In the next part of this never ending tutorial, we will start looking at the web client side of our component. We will start gently (after this overdose of code ☺) by looking at how we can build components that will not break in the web client, and then by creating a simple Wicket label component to hold our value.

Later, in again another part, we will implement the full Wicket slider equivalent of our Swing slider for the web client.

So stay tuned!

In the meantime, have fun with sliders galore in the smart client ☺!

Patrick Talbot
Servoy Stuff
2009-07-08