Creating New Controls

In the previous section, you learned how to compose complex controls out of existing controls. This chapter now will tell you how to build completely new controls which are not yet part of the Application Designer control set. The following topics are covered:

- Concept
- Example 1
- JavaScript Functions
- Example 2
- Example 3 (Applet)
- Summary

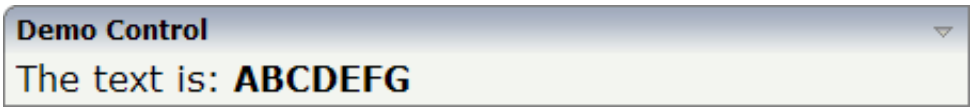
Concept

The concept of building your own controls is to insert corresponding HTML and JavaScript instructions into the HTML page which is the result of the generation process.

There is a JavaScript function library available which can be directly accessed inside the HTML code which is generated. This library contains very useful methods for accessing properties and executing methods of the "model" ("net data") behind the page.

Example 1

The first example is a quite simple one: a tag with the name "democontrol" is introduced, which does nothing else than writing a text which is passed via a tag attribute into the generated HTML page:



The corresponding XML layout definition looks as follows:

```
<rowarea name="Demo Control">
    <itr>
        <demo:democontrol text="ABCDEFG">
        </demo:democontrol>
    </itr>
</rowarea>
```

You see that the text which is passed inside the `text` attribute of the `demo:democontrol` tag is displayed inside the control in bold letters.

The Java code of the tag handler of the `demo:democontrol` tag looks as follows:

```
package com.softwareag.cis.demolibrary;

import org.xml.sax.AttributeList;

import com.softwareag.cis.gui.generate.*;
import com.softwareag.cis.gui.protocol.*;

public class DEMOCONTROLHandler implements ITagHandler
{
    // -------------------------------------------------------------------------
    // members
    // -------------------------------------------------------------------------

    String m_text;

    // -------------------------------------------------------------------------
    // public methods
    // -------------------------------------------------------------------------

    /**
     */
    public void generateHTMLForStartTag(
        int id,
        String tagName,
        AttributeList attrlist,
        ITagHandler[] handlersAbove,
        StringBuffer sb,
        ProtocolItem protocolItem)
    {
        readAttributes(attrlist);
        fillProtocolItem(protocolItem);
        sb.append("\n<!-- DEMOCONTROL begin -->\n");
        sb.append("<td>The text is: <b>"+m_text+"</b></td>\n");
    }

    /**
     */
    public void generateHTMLForEndTag(String tagName, StringBuffer sb)
    {
        sb.append("\n<!-- DEMOCONTROL end -->\n");
    }

    /**
     */
    public void generateJavaScriptForInit(
        int id,
        String tagName,
        StringBuffer sb)
    {
    }

    // -------------------------------------------------------------------------
    // private methods
    // -------------------------------------------------------------------------

    /**
     */
    public void readAttributes(AttributeList attrlist)
    {
        for (int i=0; i<attrlist.getLength(); i++)
        {
            if (attrlist.getName(i).equals("text"))
```

```
                m_text = attrlist.getValue(i);
            }
        }

        /**
         */
        public void fillProtocolItem(ProtocolItem pi)
        {
            if (m_text == null)
                pi.addMessage(new Message(Message.TYPE_ERROR,
                                          "Attribute TEXT is not defined"));
        }

}
```

In the tag handler, the following steps are processed:

- In the `generateHTMLForStartTag()` method, the attributes which are defined with the tag are read first and the protocol is filled.
- Then, plain HTML information is appended to the HTML string which is passed as a parameter (StringBuffer `sb`). Inside the HTML information, the value of the `text` attribute is dynamically inserted.

This control does not provide for any interactivity; it just writes out a certain value which is defined inside its tag definition.

JavaScript Functions

For more interactive controls - for example, which use certain data coming from the server side adapter - you need to access certain JavaScript functions which are available inside the client. Inside the HTML page which is generated, there is an object "csciframe" available which provides for a certain set of functions to use.

It is not possible in JavaScript to arrange a set of published functions in some kind of interface in order to only allow users a dedicated access. Therefore, the functions which are allowed to access are listed in this section. You must only stay with these functions - even if you may see additional ones in the JavaScript sources of Application Designer.

Function	Description

setPropertyValue(pn,pv)	Sets a property value inside the adapter. The value is not directly sent to the server but is buffered first in the client. If there is a synchronization event, then the buffer is transferred.
	
	pn = name of property
	
	pb = value
	
	Examples:
	
	`csciframe.setPropertyValue(companyName,"Software AG");`
	
	`csciframe.setPropertyValue(address.firstName,"John");`
	
	`csciframe.setPropertyValue(addresses[2].firstName,"Maria");`

Function	Description

getPropertyValue(pn)	Reads a property value from the adapter (better: the client representation of the adapter).
	
	pn = name of property
	
	result = string of property value
	
	Examples:
	
	`var vResult1 = csciframe.getPropertyValue("company");`
	
	`var vResult2 = csciframe.getPropertyValue("addresses[2].firstName");`
	
	Pay attention: the adapter value is always passed back as a string.
	
	A boolean value, as a consequence, is returned as "true" string and not as "true" boolean value.
	
	Null values of the adapter, i.e. where the Java adapter class on the server side passes back "null", are returned as an empty string ("").
	
	A JavaScript null value is passed back if the property for which you ask does not exist.

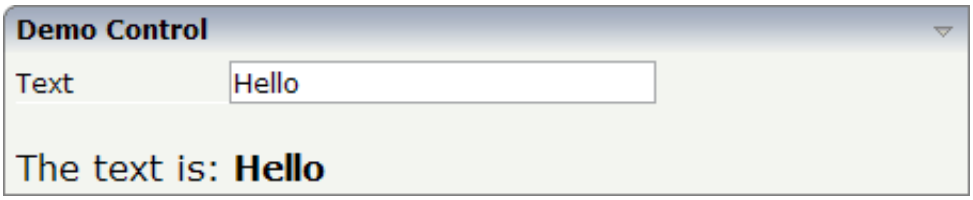
registerListener(me)	Passes a method pointer (me value). The method is called every time when a response of a client request is processed. In other words: every time new data comes from the server or if the model is updated in another way (e.g. by flush signals of other controls), then the corresponding methods are called. In the method, you can place a corresponding reaction of your control on new data.
	
	The method which you pass must have a parameter `model` - which is not used anymore, but which has to be defined.
	
	Example:
	
	`...`
	`...`
	`function reactOnNewData(model)`
	`{`
	`  var vResult = csciframe.getPropertyValue("firstName");`
	`  alert(vResult);`
	`}`
	`...`
	`...`
	`csciframe.registerListener(reactOnNewData);`
	`...`
	`...`

invokeMethodInModel(mn)	Invokes the calling of a method inside the adapter. As a consequence, the data changes which may have been buffered inside the client are flushed to the server and the method is called. mn = name of adapter method Example: `csciframe.invokeMethodInModel("onSave");`

Function	Description
submitModel(n)	Synchronizes the client with the server. Analogous to the `invokeMethodInModel()` method from the synchronization point of view - but now without calling an explicit method in the adapter. n = name, must be `submit` Example: `csciframe.submitModel("submit");`

Example 2

The following example is an extension of the previous example. Whereas in *Example 1* the text which is output by the control was defined as an attribute of the tag definition, the text is now dynamically derived from an adapter property.



The XML layout definition is:

```
<rowarea name="Demo Control">
    <itr>
        <label name="Text" width="100">
        </label>
        <field valueprop="text" width="200" flush="screen">
        </field>
    </itr>
    <vdist height="20">
    </vdist>
    <itr>
        <demo:democontroldyn textprop="text">
        </demo:democontroldyn>
    </itr>
</rowarea>
```

You see that the DEMOCONTROLDYN control references the same adapter property `text` as the FIELD control.

Let us have a look at the tag handler class for the DEMOCONTROLDYN control:

```
package com.softwareag.cis.demolibrary;

import org.xml.sax.AttributeList;

import com.softwareag.cis.gui.generate.*;
import com.softwareag.cis.gui.protocol.*;

public class DEMOCONTROLDYNHandler implements ITagHandler
{
    // -------------------------------------------------------------------------
    // members
```

```
// ----------------------------------------------------------------------

String m_textprop;

// ----------------------------------------------------------------------
// public methods
// ----------------------------------------------------------------------

/**
 */
public void generateHTMLForStartTag(
    int id,
    String tagName,
    AttributeList attrlist,
    ITagHandler[] handlersAbove,
    StringBuffer sb,
    ProtocolItem protocolItem)
{
    readAttributes(attrlist);
    fillProtocolItem(protocolItem);
    sb.append("\n<!-- DEMOCONTROL begin -->\n");
    sb.append("<td>The text is: <b>");
    sb.append("<span id='DEMOSPAN"+id+"'></span>");
    sb.append("</b></td>\n");
    sb.append("<script>\n");
    sb.append("function reactOnModelUpdate"+id+"(model)\n");
    sb.append("{\n");
    sb.append("  var vText = csciframe.getPropertyValue('"+m_textprop+"');\n");
    sb.append("  var vSpan = document.getElementById('DEMOSPAN"+id+"');\n");
    sb.append("  vSpan.innerHTML = vText;\n");
    sb.append("}\n");
    sb.append("</script>\n");
}

/**
 */
public void generateHTMLForEndTag(String tagName, StringBuffer sb)
{
    sb.append("\n<!-- DEMOCONTROL end -->\n");
}

/**
 */
public void generateJavaScriptForInit(
    int id,
    String tagName,
    StringBuffer sb)
{
    sb.append("csciframe.registerListener(reactOnModelUpdate"+id+");\n");
}

// ----------------------------------------------------------------------
// private methods
// ----------------------------------------------------------------------

/**
 */
public void readAttributes(AttributeList attrlist)
{
    for (int i=0; i<attrlist.getLength(); i++)
    {
        if (attrlist.getName(i).equals("textprop"))
```

```
                m_textprop = attrlist.getValue(i);
            }
        }

        /**
         */
        public void fillProtocolItem(ProtocolItem pi)
        {
            // Messages
            if (m_textprop == null)
                pi.addMessage(new Message(Message.TYPE_ERROR,
                                          "Attribute TEXTPROP is not defined"));
            // Property Usage
            pi.addProperty(m_textprop,"String");
        }

}
```

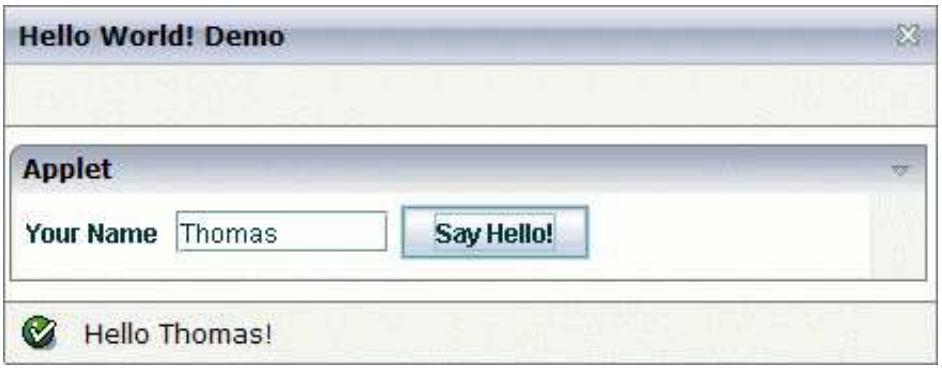
You see:

- Inside the `generateJavaScript()` method, a JavaScript function is added as a listener to adapter model changes. The function is generated inside the `generateHTMLForStartTag()` method.
- The name of the property is read via the attribute list into the member `m_textprop` - and is dynamically used when calling the JavaScript function `getPropertyValue()`.
- All JavaScript names (e.g. method names, IDs of controls) which are "global" inside the HTML page are suffixes with the control ID which is passed via the `ITagHandler` methods. The reason: if one control is defined multiple times inside a page, then the different methods and IDs are separated by this ID.
- The protocol item is filled with the information about the property which is required by the control. This is necessary at runtime because the Application Designer runtime environment needs to find out which data of an adapter property to send back to the client (see *Binding between Page and Adapter* in the *Special Development Topics*).

Example 3 (Applet)

This example shows how to embed Java applets. Unlike *Example 1* and *Example 2*, the generated HTML/JavaScript here just makes the applet aware of the normal data communication between screen and page adapter. The rendering is done in Java.

This example uses a very basic "Say Hello!" applet. It shows one field input and a button. On button click, a message is displayed on the STATUSBAR control.



The XML layout definition is:

```
<page model="SayHelloAdapter">
    <titlebar name="Say Hello! Demo">
    </titlebar>
    <header>
    </header>
    <pagebody>
        <rowarea name="Applet">
            <tr>
                <demo:applet code="SayHelloApplet.class"
                             width="400"
                             height="40"
                             valueprop="yourName"
                             method="onSayHello">
                </demo:applet>
            </tr>
        </rowarea>
    </pagebody>
    <statusbar>
    </statusbar>
</page>
```

The `demo:applet` tag shows the usual applet attributes: `code`, `width` and `height`. With the attribute `valueprop`, the applet's field input is bound to the adapter property `yourName`. The attribute `method` binds the button to adapter method `onSayHello`.

This is the page adapter for this example:

```
import com.softwareag.cis.server.Adapter;

public class SayHelloAdapter extends Adapter
{
    // property >yourName<
    String m_yourName;
    public String getYourName() { return m_yourName; }
    public void setYourName(String value) { m_yourName = value; }

    /** called on button click */
    public void onSayHello()
    {
      outputMessage(MT_SUCCESS, "Hello "+m_yourName+"!");
    }
}
```

The adapter only provides for the property `yourName` and the method `onSayHello`.

The most important thing is the tag handler class. Let us have a look at `APPLETHandler`:

```
package com.softwareag.cis.demolibrary;

import org.xml.sax.*;

import com.softwareag.cis.file.CSVManager;
import com.softwareag.cis.gui.protocol.*;
import com.softwareag.cis.gui.util.*;

public class APPLETHandler implements ITagHandler
{
    // -------------------------------------------------------------------------
    // members
    // -------------------------------------------------------------------------

    String m_code;
    String m_codebase = ".";
    String m_width = "100";
    String m_height = "100";
    String m_valueprop;
    String m_method;

    // -------------------------------------------------------------------------
    // public usage
    // -------------------------------------------------------------------------

    public void generateHTMLForStartTag(int id,
                                        String tagName,
                                        AttributeList attrlist,
                                        ITagHandler[] handlersAbove,
                                        StringBuffer sb,
                                        ProtocolItem protocolItem)
    {
        readAttributes(attrlist);
        fillProtocol(protocolItem);

        sb.append("\n");
        sb.append("<!-- APPLET begin -->\n");
        sb.append("<td>\n");
        sb.append("<applet name=\"APPLET"+id+"\" " +
                                "codebase=\".\" " +
                                "code=\""+m_code+"\" " +
                                "width=\""+m_width+"\" " +
                                "height=\""+m_height+"\" " +
                                "MAYSCRIPT>\n");
        sb.append("   <param name=\"scriptable\" value=\"true\">\n");
        sb.append("   <param name=\"id\" value=\""+id+"\">\n");
        sb.append("   <param name=\"valueprop\" value=\""+m_valueprop+"\">\n");
        sb.append("   <param name=\"method\" value=\""+m_method+"\">\n");
        sb.append("</applet>\n");

        sb.append("<script>\n");
        sb.append("function romu"+id+"(model) {");
        sb.append("    "try { document.APPLET"+id+".reactOnNewData(); } \n");
        sb.append("    "catch (exc) { alert('Error occurred when talking to applet!' + exc); }\n");
        sb.append("} \n");
        sb.append("function getProperyValue"+id+"(propertyName) { return C.getPropertyValue(propertyName); } \n");
        sb.append("function setPropertyValue"+id+"(propertyName, value) { C.setPropertyValue(propertyName, value); } \n");
        sb.append("function invokeMethodInModel"+id+"(methodName) { C.invokeMethodInModel(methodName); } \n");
        sb.append("</script>\n");
    }

    public void generateHTMLForEndTag(String tagName,
                                      StringBuffer sb)
    {
        sb.append("<!-- APPLET end -->\n");
        sb.append("</td>\n");
    }

    public void generateJavaScriptForInit(int id,
                                          String tagName,
                                          StringBuffer sb)
    {
        sb.append("C.registerListener(romu"+id+");\n");
    }

    // -------------------------------------------------------------------------
    // private usage
    // -------------------------------------------------------------------------

    /** */
    private void readAttributes(AttributeList attrlist)
```

```
    {
        for (int i=0; i<attrlist.getLength(); i++)
        {
            if (attrlist.getName(i).equals("code")) m_code = attrlist.getValue(i);
            if (attrlist.getName(i).equals("codebase")) m_codebase = attrlist.getValue(i);
            if (attrlist.getName(i).equals("width")) m_width = attrlist.getValue(i);
            if (attrlist.getName(i).equals("height")) m_height = attrlist.getValue(i);
            if (attrlist.getName(i).equals("valueprop")) m_valueprop = attrlist.getValue(i);
            if (attrlist.getName(i).equals("method")) m_method = attrlist.getValue(i);
        }
    }

    /** */
    private void fillProtocol(ProtocolItem pi)
    {
        // check
        if (m_code == null) pi.addMessage(new Message(Message.TYPE_ERROR, "CODE not set"));
        if (m_valueprop == null) pi.addMessage(new Message(Message.TYPE_ERROR, "VALUEPROP not set"));
        if (m_method == null) pi.addMessage(new Message(Message.TYPE_ERROR, "METHOD not set"));
    }

}
```

You see:

- `readAttributes()` reads all attributes from the XML. Their values are saved in member variables.
- `fillProtocol()` checks whether mandatory attributes are set. If not, an error message is shown within the generation protocol.
- `generateHTMLForStartTag()` generates HTML code containing the applet/parameter tags. Some JavaScript functions are added that are available inside the applet coding (`getPropertyValue`/`setPropertyValue`/`invokeMethodInModel`).
- `generateJavaScriptForInit()` registers function `romu()` as a listener to model changes. On change, the applet is called by `reactOnNewData`.

The Java applet looks as follows:

```
import netscape.javascript.JSObject;

public class SayHelloApplet extends Applet
{
      private String m_id;
      private String m_valueprop;
      private String m_method;
      private JLabel m_label;
      private JTextField m_fieldJ;
      private JButton m_buttonJ;
      private JSObject m_windowJ = null;

    // ------------------------------------------------------------------------
    // Data binding
    // ------------------------------------------------------------------------

      /**
       * Callback method for model change events
       */
      public void reactOnNewData()
      {
            Object[] args = new String[] { m_valueprop };
            Object v = m_windowJ.call("getProperyValue"+m_id, args);
            if (v == null)
                  v = "";
            m_fieldJ.setText((String)v);
```

```
      }

      /**
       * button action handler
       */
      private void buttonAction()
      {
            // pass field's value into property
            Object[] args = new String[] { m_valueprop , m_fieldJ.getText() };
            m_windowJ.call("setPropertyValue"+m_id, args);

            // call adapter method
            args = new String[] { m_method };
            m_windowJ.call("invokeMethodInModel"+m_id, args);
      }

      /**
   * Is called on load
   */
      public void init()
      {
            m_windowJ = JSObject.getWindow(this);
            m_id = getParameter("id");
            m_valueprop = getParameter("valueprop");
            m_method = getParameter("method");

            createGUI();
      }

    // ------------------------------------------------------------------------
    // applet specific methods
    // ------------------------------------------------------------------------

      private void createGUI(){..}

      public void destroy(){..}

      private void cleanUp(){..}
}
```

You see:

- JavaScript methods are called using `JSObject`. It is part of *plugin.jar* of Sun's Java Virtual machine.
- The method `reactOnNewData()` accesses the fresh property value.
- The method `buttonAction()` first sets the user input and then calls the adapter method.

Summary

Writing new controls requires a profound knowledge of HTML and JavaScript. In principle, everything is simple, but there are a couple of pieces which have to be put together in order to form a control properly:

- You have to render the control via HTML.

- You have to manipulate the control via JavaScript - in case you have a dynamic control.
- You have to bind the control to adapter properties/methods.
- You have to pay attention to the fact that all controls are living in the same page - and there must not be any confusion with naming of IDs and method names.
- You have to use the JavaScript initialisation for registering your control inside the internal eventing when new page content arrives inside the client.
- You have to properly fill the protocol item.

There are still some issues which are not mentioned yet - but which you might be interested when writing certain controls. The concept is covered inside this section - and after having built your first controls, there is no more complex item to be added.