GlassFish ESB v2.1 Field Notes
Using JavaScript Codelets to Extend BPEL 2.0 Functionality

Michael.Czapski@sun.com
December 2009, Release 1.0.2
Incorporating comments and improvements by Vladimir Iaroslavski, with thanks.

Table of Contents

Introduction

The BPEL SE, featured in the GlassFish ESB, the OpenESB and the Java CAPS 6, has the ability to execute JavaScript (ECMAScript) code inline. Why would one do that, you may ask. The answer is: because BPEL, great as it is with XML all over the place and all, can not do everything, and invoking Web Services and POJOs from BPEL for small and simple code adds too much overhead.

Take a date conversion, for example. It takes about 4 lines of Java code to implement this functionality. Doing this in BPEL is too horrible to contemplate. Doing this in JavaScript is not too bad, given availability of ready-made JavaScript code that do the job. The issue is that one cannot invoke Java from BPEL without resorting to a web service or a Plain Old Java Object (POJO). Invoking JavaScript, on the other hand, does not require either. Furthermore, JavaScript, in the Netscape days, acquired the ability to embed Java using technology known as LiveConnect.

This Note discusses the integration of BPEL and JavaScript and walks through the implementation of two in-line JavaScript and Java codelets. One gives the BPEL process the ability to obtain a value of a JVM Argument, used by the hosting Application Server, or a Java System Property from the environment of the Application Server. The other implements date format conversion functionality.

JavaScript and Java

In this note my interest is not so much in JavaScript itself as in the ability to embed Java statements in JavaScript.

Take the following line of code:

```
var sPropValue = java.lang.System.getProperty(sPropName);
```

`sPropValue` is a JavaScript variable. `sPropName` is a JavaScript variable. `java.lang.System.getProperty()` is a `getProperty()` method of the Java `System` object. In this case, if we manage to embed this piece of JavaScript in BPEL, we have access to values of properties in the Application Server environment without resorting to web service or POJO invocation.

Now take this block of code:

```
function cvtDate(vDateTimeIn, vDateTimeInFormat, vDateTimeOutFormat)
{
//    var vDateTimeIn = "10/12/1956";
//    var vDateTimeInFormat = "dd/MM/yyyy";
//    var vDateTimeOutFormat = "dd-MM-yyyy'T'hh:mm:ss";

  fmtIn = new java.text.SimpleDateFormat(vDateTimeInFormat);
  fmtOut = new java.text.SimpleDateFormat(vDateTimeOutFormat);
  var dt = fmtIn.parse(vDateTimeIn);
  return fmtOut.format(dt);
}

var sDateTimeOut = cvtDate
                          (vDateTimeIn
                          ,vDateTimeFmtIn
                          ,vDateTimeFmtOut);
```

This code first defines a JavaScript function `cvtDate`(), which accepts 3 arguments, does its job and returns the result. Java class `SimpelDateFormat`, in the `text` package, is used to convert date/time string from one format to another. JavaScript variable `sDateTimeOut` receives the return value of the JavaScript function `cvtDate`, which uses Java to do the work on JavaScript variables. JavaScript itself does not have built-in facilities for easy date formatting so the ability to use Java for this is a great thing.

I have not explored the depth of integration between JavaScript and Java or looked at the limitations of the method. There is plenty of information on JavaScript, which has been in use for Web scripting since early 1990s, and on LiveConnect which has been around for almost as long as JavaScript.

See the following links for additional material:

- http://devedge-temp.mozilla.org/central/javascript/index_en.html - JavaScript Central - JavaScript reference documentation
- https://developer.mozilla.org/en/Core_JavaScript_1.5_Guide - JavaScript 1.5 Guide
- https://developer.mozilla.org/En/E4X/Processing_XML_with_E4X - Processing XML with JavaScript
- http://devedge-temp.mozilla.org/library/manuals/2000/javascript/1.5/guide/lc.html#1008305 and https://developer.mozilla.org/En/Core_JavaScript_1.5_Guide:LiveConnect_O

verview – LiveConnect material – highly recommended for these who would like to use Java in JavaScript
- http://www.mozilla.org/rhino/ - The Rhino project: JavaScript for Java
- http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf - a densely formal definition of the ECMAScript (JavaScript in its standardized form)
- http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-357.pdf - a densely formal definition of the ECMAScript for XML (E4X)

In addition, there are plenty of more or less useful JavaScript functions in freely available script libraries and descriptions of useful functions in a plethora of Internet-borne articles and posts.

Developing JavaScript with embedded Java

The develop-deploy-debug cycle for Enterprise applications is fairly time consuming. This can get quite irritating when trying to work iteratively with dynamic languages embedded in non-dynamic environment. It is far better to develop and debug JavaScript outside BPEL. There are tools that facilitate this work. Mozilla Rhino's JavaScript Shell (http://www.mozilla.org/rhino/shell.html), for example, allows one to execute a JavaScript code in a file or to execute JavaScript statements interactively.

Download the JavaScript Shell and "install" it for this exercise.

Create the following JavaScript code in a file called CvtDateTime.js

```
// begin JavaScript code
//
var vDateTimeIn = arguments[0];
var vDateTimeInFormat = arguments[1];
var vDateTimeOutFormat = arguments[2];

function cvtDateTime(vDateTimeIn, vDateTimeInFormat, vDateTimeOutFormat) {
//     var vDateTimeIn = "10/12/1956";
//     var vDateTimeInFormat = "dd/MM/yyyy";
//     var vDateTimeOutFormat = "dd-MM-yyyy'T'hh:mm:ss";
  fmtIn = new java.text.SimpleDateFormat(vDateTimeInFormat);
  fmtOut = new java.text.SimpleDateFormat(vDateTimeOutFormat);
  var dt = fmtIn.parse(vDateTimeIn);
  return fmtOut.format(dt);
}

var sDateTimeOut = cvtDateTime
        (vDateTimeIn
        ,vDateTimeInFormat
        ,vDateTimeOutFormat);

java.lang.System.out.println("Converted Date: " + sDateTimeOut);

//
// end JavaScript code
```

Run the JavaScript shell, submitting the cvtDateTime.js script with the appropriate arguments. This is what I saw when I did this:

```
%JAVA_HOME%\bin\java -cp c:\tools\rhino1_7R2\js.jar org.mozilla.javascrip
t.tools.shell.Main cvtDateTime.js "10/11/1927" "dd/MM/yyyy" "yyyy-MM-dd'T
'hh:mm:ss"
Converted Date: 1927-11-10T12:00:00
```

It is easy to modify the script and re-execute it until it is correct.

Note that I am using `java.lang.System.println()` method to emit debugging information to the standard output. In JavaScript Shell I could have used the JavaScript `print()` statement instead. This would work for the JavaScript Shell but it would not work for the JavaScript embedded in BPEL. Since this is where my scripts will ultimately run I am using the facilities I know will work in the target environment.

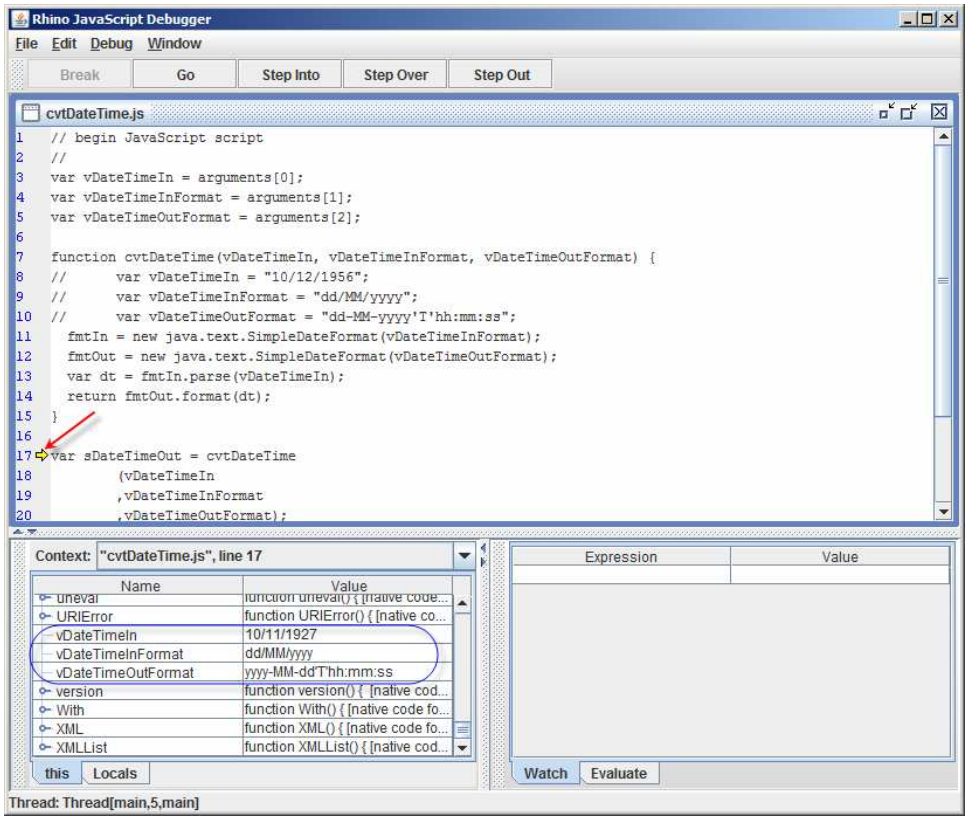
The Shell is fine for simple scripts and for people who are very familiar with the languages involved. For heavy duty script development a JavaScript Debugger will probably be a good thing. Fortunately, there is the Rhino JavaScript Debugger, http://www.mozilla.org/rhino/debugger.html. I am not going to dive deeply into its operation. See the on-line material on the site for this.

Assuming the Rhino JavaScript Debugger has been downloaded and "installed", we can exercise the cvtDateTime.js in the debugger.

For me the command line is:

```
%JAVA_HOME%\bin\java -cp c:\tools\rhino1_7R2\js.jar org.mozilla.javascrip
t.tools.debugger.Main cvtDateTime.js "10/11/1927" "dd/MM/yyyy" "yyyy-MM-d
d'T'hh:mm:ss"
```

Here is a screenshot of the Debugger window at the point the cvtDateTime function is about to be invoked, showing values of the arguments given on the command line.



Note that I am starting the script with:

```
var vDateTimeIn = arguments[0];
var vDateTimeInFormat = arguments[1];
var vDateTimeOutFormat = arguments[2];
```

This is to provide values to use in the script in the stand-alone environment and is specific to the Mozilla Java Script Shell and the Mozilla JavaScript Debugger. When invoking this script from BPEL we will provide values in a different way and will return the result to BPEL rather then dumping it to the standard output. This will be discussed later. For now we have a functioning JavaScript code which uses Java classes to convert a date/time value from input format to output format. We can use the body of this script in BPEL.

Let's now introduce a different JavaScript code, one which returns the value of a JVM argument or the Java system property – GetPropertyValue.js.

```
// begin JavaScript code
//
var sPropName = arguments[0];

var sPropValue = java.lang.System.getProperty(sPropName);

java.lang.System.out.println("Property Value: " + sPropValue);

//
// end JavaScript code
```

Here is what I do and see when executing this script using the JavaScript Shell:

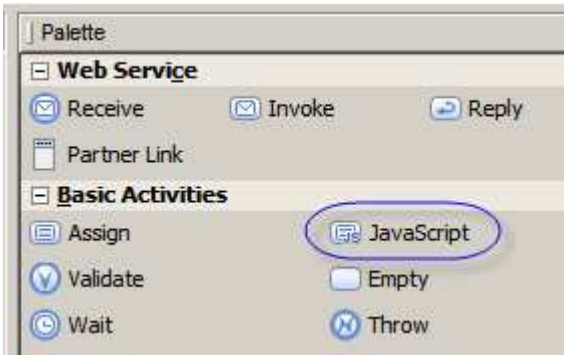
```
%JAVA_HOME%\bin\java -cp c:\tools\rhino1_7R2\js.jar org.mozilla.
javascript.tools.shell.Main GetPropertyValue.js "user.home"
Property Value: C:\Documents and Settings\mczapski
```

Using this script in this manner gives us access to the values of system properties and JVM arguments in the JVM used by the JavaScript Shell. Embedding this kind of script in BPEL will give us access to named system properties and JVM arguments visible to the Application Server in which my BPEL SU executes. We will use this script later to determine the host name where the BPEL SU executes.

Other useful functionality can be developed either in pure JavaScript or in combination of JavaScript and Java.

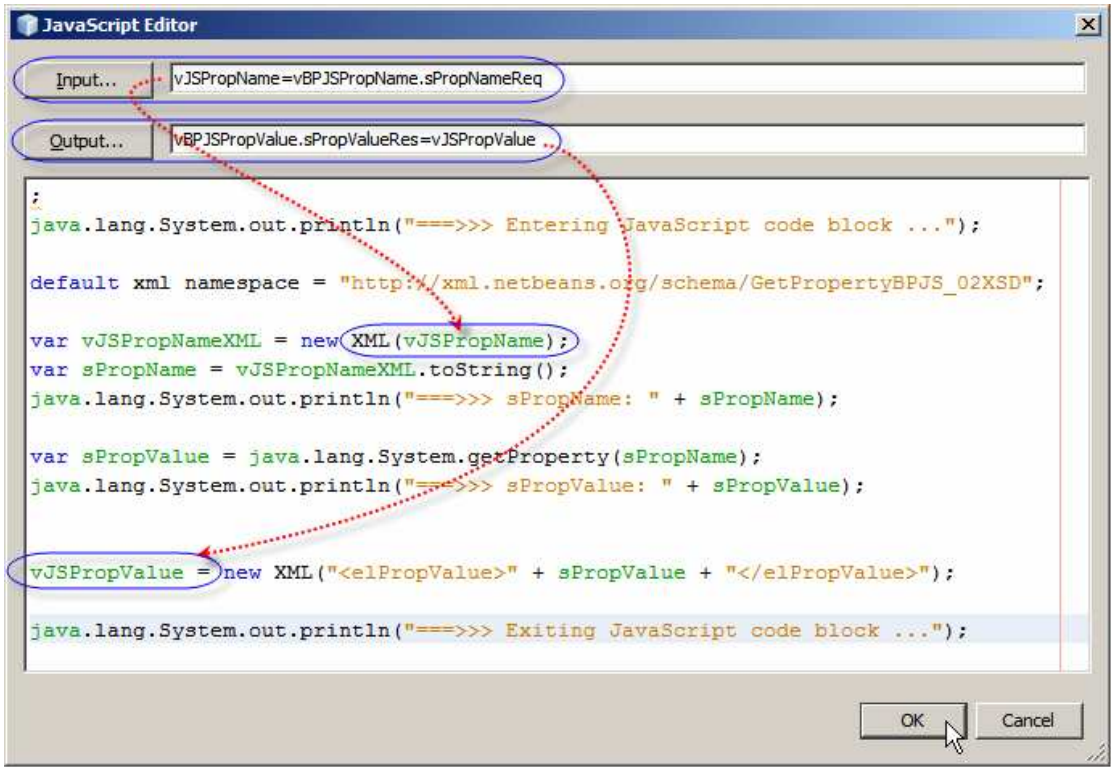
Embedding JavaScript in BPEL

BPEL 2.0 Editor Palette has a JavaScript Activity which would be added to the process model to embed the JavaScript code.



Embedded JavaScript code does not have direct access to variables defined in BPEL. The JavaScript Editor allows one to name JavaScript variables and associate them with BPEL variables for input and output. The underlying infrastructure will move values between BPEL variables and JavaScript variables.

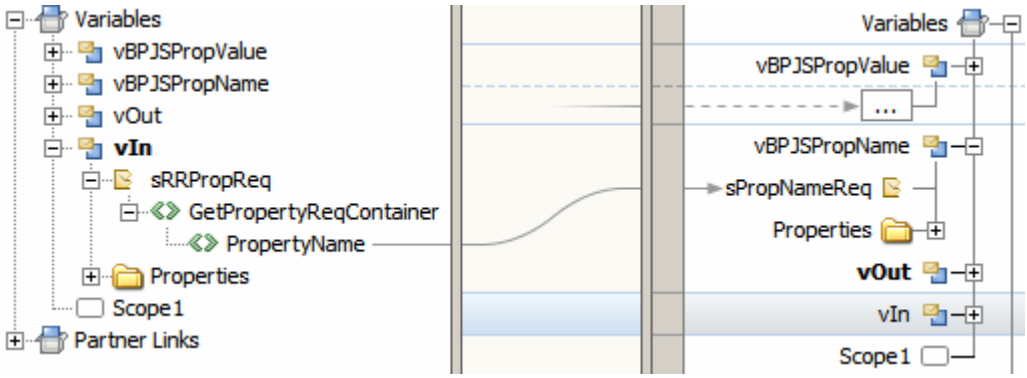
The following figure, showing the JavaScript Editor, highlights the JavaScript variables used as intermediaries between BPEL and JavaScript.



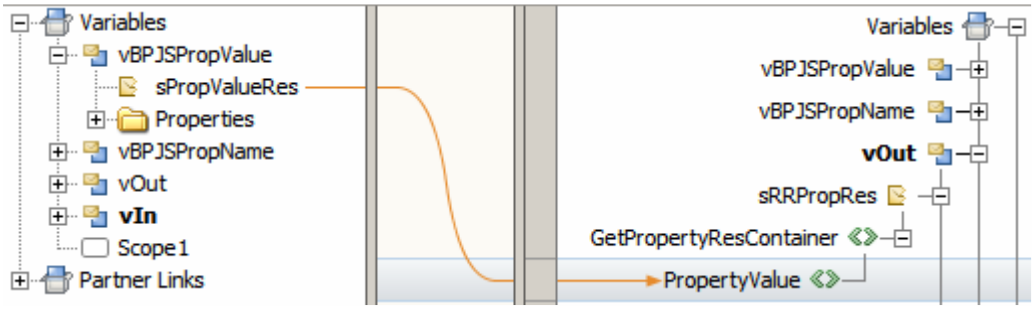
Note the BPEL variable references, `vBPJSPropName.sPropNameReq` and `vBPJSPropValue.sPropValueRes`. Values assigned to the variable `vBPJSPropName.sPropNameReq` in BPEL will be visible to the JavaScript through the JavaScript variable `vJSPropName`. Values assigned to JavaScript variable `vJSPropValue` will be visible to BPEL through the BPEL variable `vBPJSPropValue.sPropValueRes`.

BPEL process, around the JavaScript activity, might look like this:

Assign2 Activity, in which the `vBPJSPropName` BPEL variable is populated, might look like this in BPEL Mapper:



Assign3 Activity, in which `vBPJSPropValue` variable value is used in assignment, might look like this the BPEL Mapper:



In summary, whatever gets assigned in BPEL to `vBPJSPropName` before the JavaScript Activity, is accessible to JavaScript through JavaScript variable `vJSPropName` (`vJSPropName=vBPJSPropName.sPropNameReq`). Whatever gets assigned to JavaScript variable `vJSPropValue` in JavaScript Activity is accessible to BPEL through BPEL variable `vBPJSPropValue` (`vBPJSPropValue.sPropValueRes=vJSPropValue`).

```
BPEL
      vBPJSPropName.sPropNameReq -> vJSPropName
            JavaScript code
                  use vJSPropName
                  assign vJSPropValue
      vJSPropValue -> vBPJSPropValue.sPropValueRes
BPEL
```

There are a couple of rules that must be followed to successfully embed JavaScript in BPEL.

Rule Number 1:

BPEL to JavaScript Interface variables must be a ***messageType*** variable.

Right now the only way to get a messageType variable is to create a WSDL with the desired XML structures as input and output messages to an operation. One can use the messages associated with the operation that triggered the process or with the operation of a web service being invoked from the process. Since all that is needed are the messages, the WSDL can be an Abstract WSDL. Such a WSDL will not be used for an endpoint and the operation will never get invoked. Defining an Abstract WSDL to get messageType variables seems like overkill but it is the only way to avoid using messages associated with Binding Components.

Rule Number 2:

BPEL variable which is to receive return value from JavaScript must be initialized before JavaScript is invoked.

Normally, a messageType BPEL variable is one in which data is received from the BC that triggered the process, which is used to return data for a request/reply service or which passes data to a BC being invoked from the process. Both of these kinds of BPEL variables are initialized / instantiated on entry. In contrast, messages defined in an Abstract WSDL which is not associated with a BC, as suggested in Rule Number 1, must be explicitly initialized before being used in JavaScript. The variable used to pass values to JavaScript is initialized with the values to be passed as a matter of course. The variable used to receive the result form JavaScript must also be initialized before invocation. If this is not done NPEs will result. Assigning empty string to leaf nodes of the variable will normally cause initialization to happen.

(Non-)**Rule Number 3:**

Use a single structure as input and a single structure as output.

This is a non-rule, in that if not followed no harm will be done.
One can specify more then one input and output variable associations, for example input: "`vBPVar2=vJSVar1,vBPVar2=vJSvar2,`…" and output: "`vJSVar1=vBPVar1, vJSVar2=vBPVar2`,…", when it is necessary to pass multiple values in or receive multiple values as results. I don't like this kind of thing for two reasons. The first is that this requires me to define more messages in the Abstract WSDL and have more messages in the mapper to clutter the display, which I don't like. The second is that I have to define a structured message, based on an XML schema, anyway. I might as well define a schema that includes all the elements I need.

Example 1: GetProperty

Let's implement the example in which we obtain the name of the host on which the BPEL SU executes. The "schematic" of the process is:

```
BPEL
       vBPJSPropName.sPropNameReq -> vJSPropName
              JavaScript code
                     use vJSPropName
                     assign vJSPropValue
       vJSPropValue -> vBPJSPropValue.sPropValueRes
BPEL
```

The BPEL process will be exposed as a request/reply SOAP Web Service to make it easy to test.

We will implement this process in two stages. In Stage I we will create the basic process which returns its input as its output. This will give us the baseline which to exercise and later modify.

Stage I

Create a BPEL Module project, JSGetPropValue.
Create a XML Schema document, AString, representing input and the output message structures.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://xml.netbeans.org/schema/AString"
    xmlns:tns="http://xml.netbeans.org/schema/AString"
    elementFormDefault="qualified">
    <xsd:element name="AStringRoot">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="StringValue" type="xsd:string"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Create a WSDL document, **JSGetPropValueIF.wsdl** as follows:

WSDL Type: **Concrete WSDL Document**
Binding: **SOAP**
Type: **Document Literal**
Operating Name: **GetPropValue**
Input:
Message Part Name: **sPropName**
Element Or Type: **AStringRoot**
Output:
Message Part Name: **sPropValue**
Element Or Type: **AStringRoot**

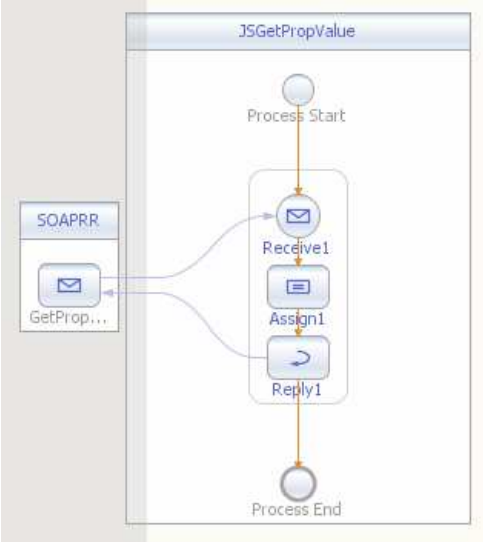
As discussed in Rule Number 1, create an Abstract WSDL, JSGetPropValueJSIFAbs, as follows:

WSDL Type: Abstract WSDL Document

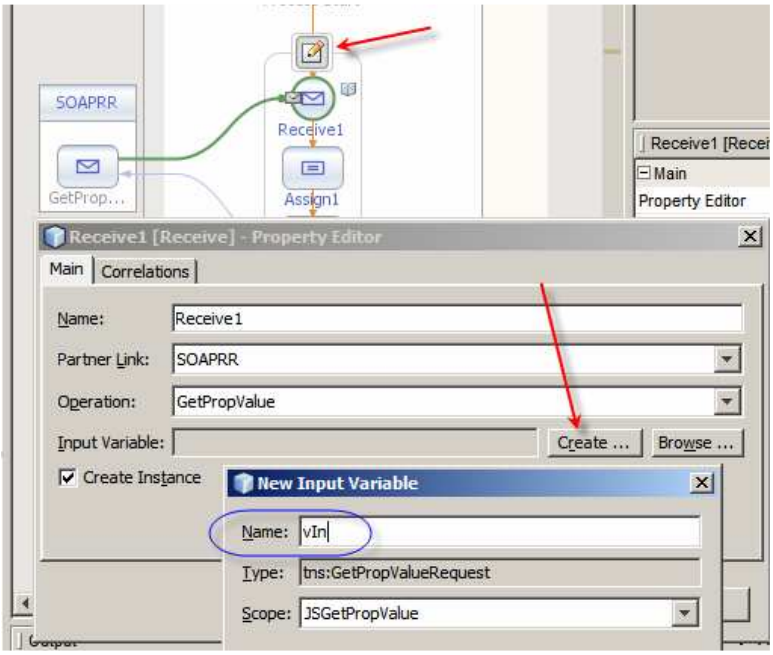
Operation Name: **BPJSInterface**
Input:
Message Part Name: **sBPJSPropNameReq**
Element Or Type: **AStringRoot**
Output:
Message Part Name: **sBPJSPropValueRes**
Element Or Type: **AStringRoot**

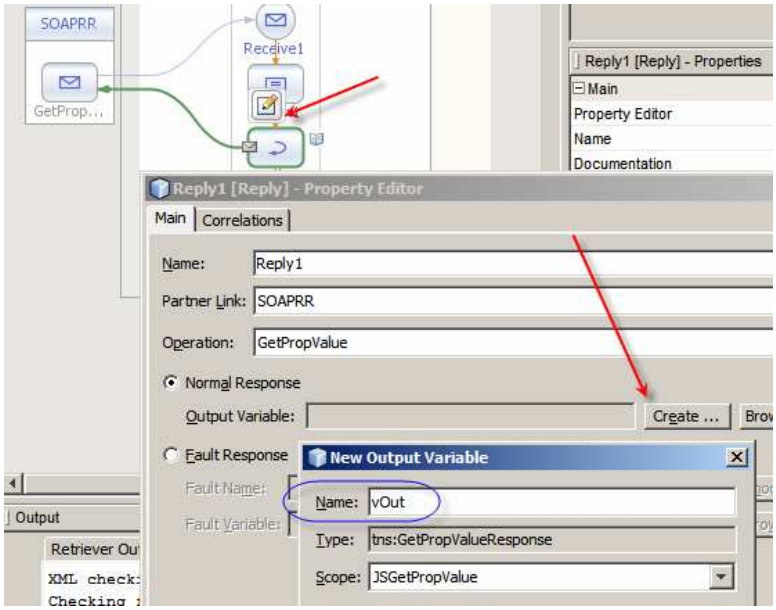
Drag the JSGetPropValueIF WSDL onto the left hand side swim line and name the partner link SOAPRR.

Add Receive, Assign and Reply activities and connect to the SOAPRR partner link.

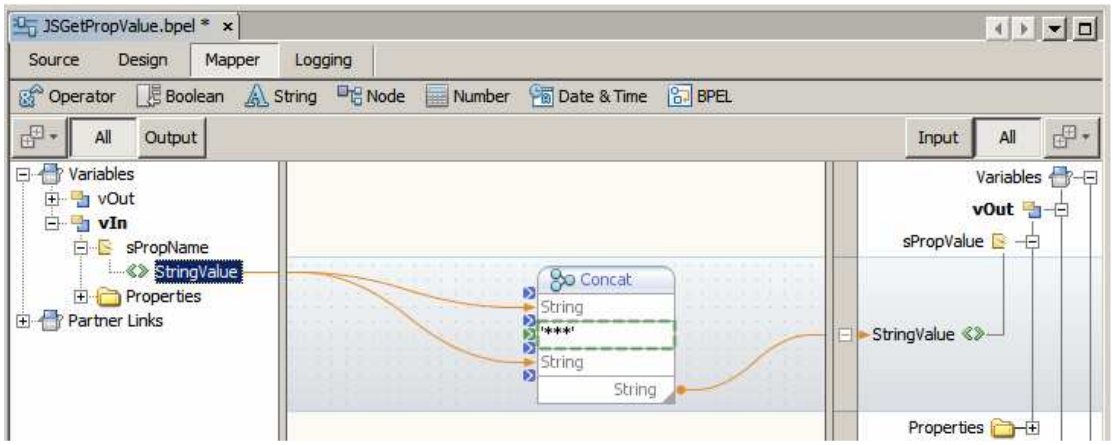


Create a variable for the Receive and name it vIn.
Create a variable for the Reply and name it vOut.





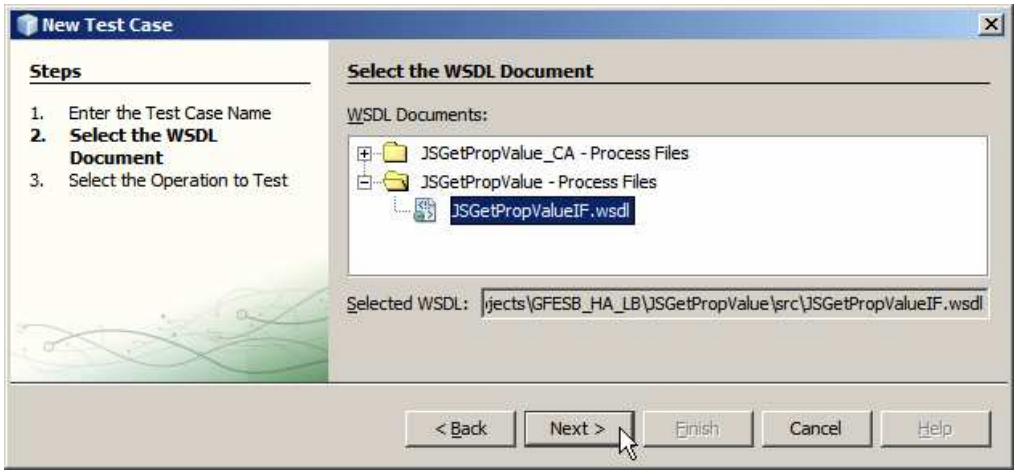
Select the Assign1 activity, switch to BPEL Mapper and map as shown:



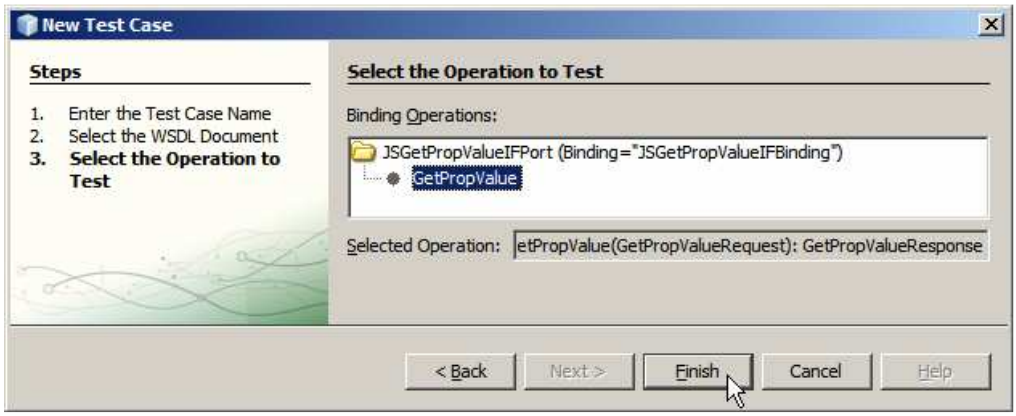
Build the project.

Create a new Composite Application project, JSGetPropValue_CA, drag the JSGetPropValue BPEL Module onto the CASA canvas, Build and Deploy.

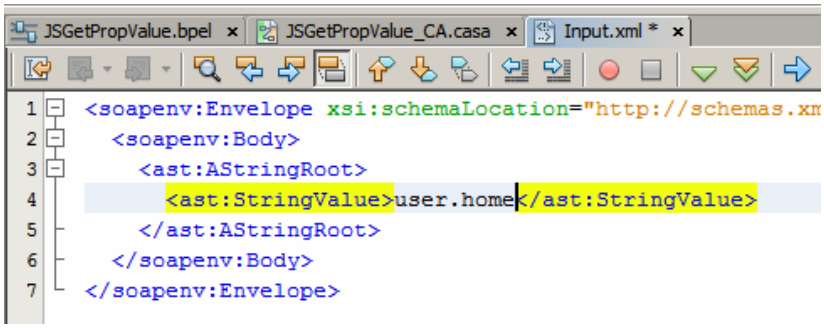
In the JSGetPropValue_CA project right click Test node and choose New Test Case. Accept default name. “Expand JSGetPropValue – Process Files” and select JSGetPropValueIF.wsdl node.



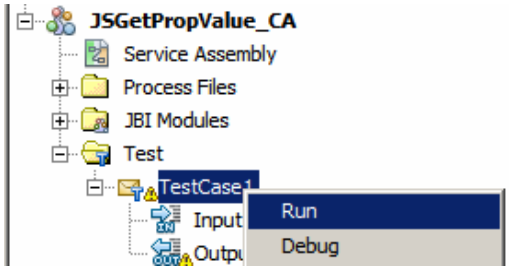
Select getPropValue operation and click Finish.



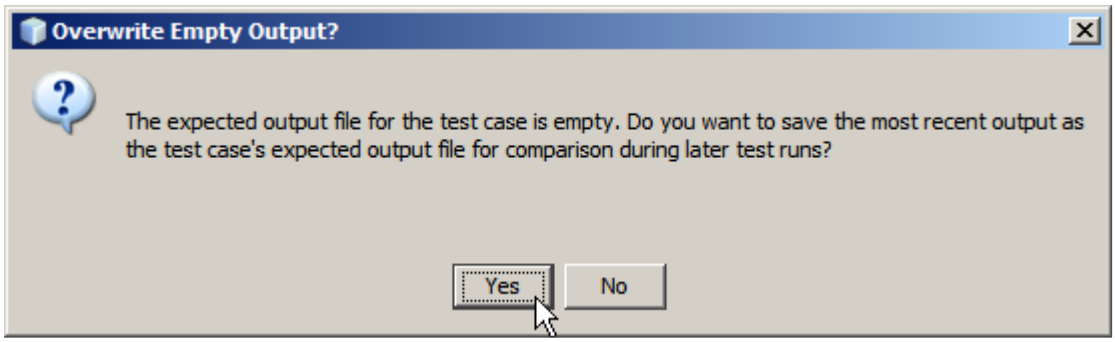
Replace `?String?` with `user.home`. This is the property whose value we are looking for.



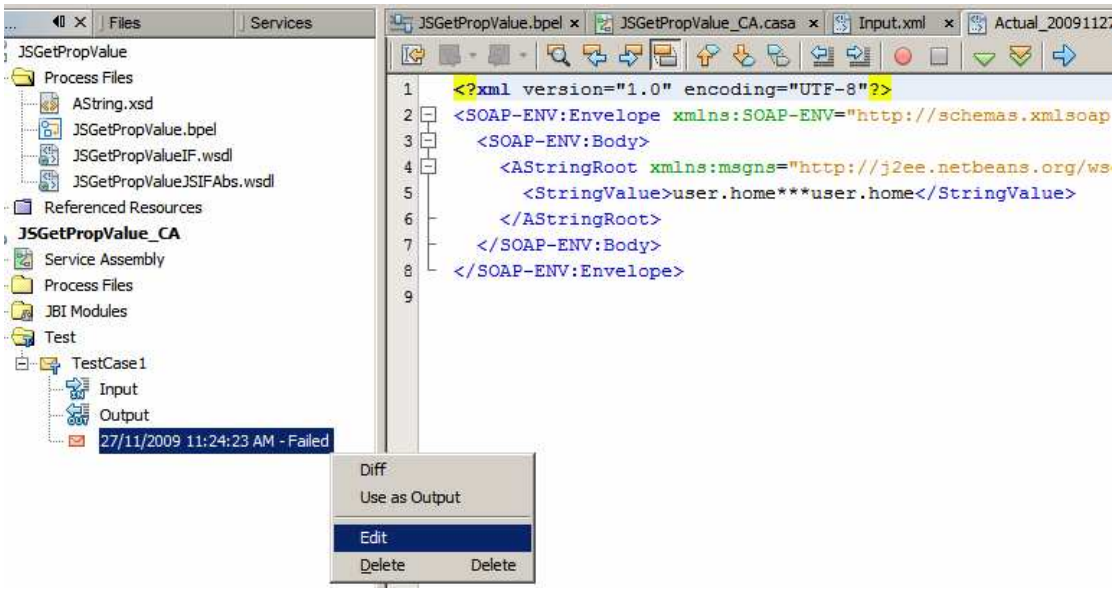
Right-click TestCase1 and choose Run.



Say “yes” to “Overwrite empty output”.



Right-click the name of the test run results, choose Edit and inspect the result.

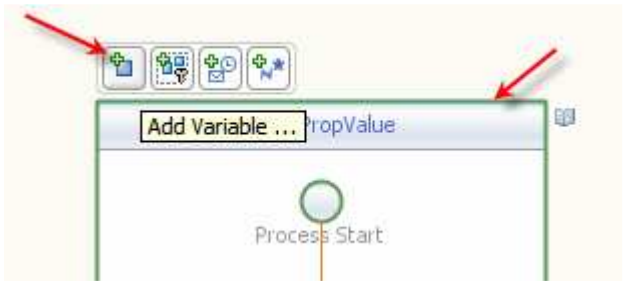


As expected, we have user.home***user.home.

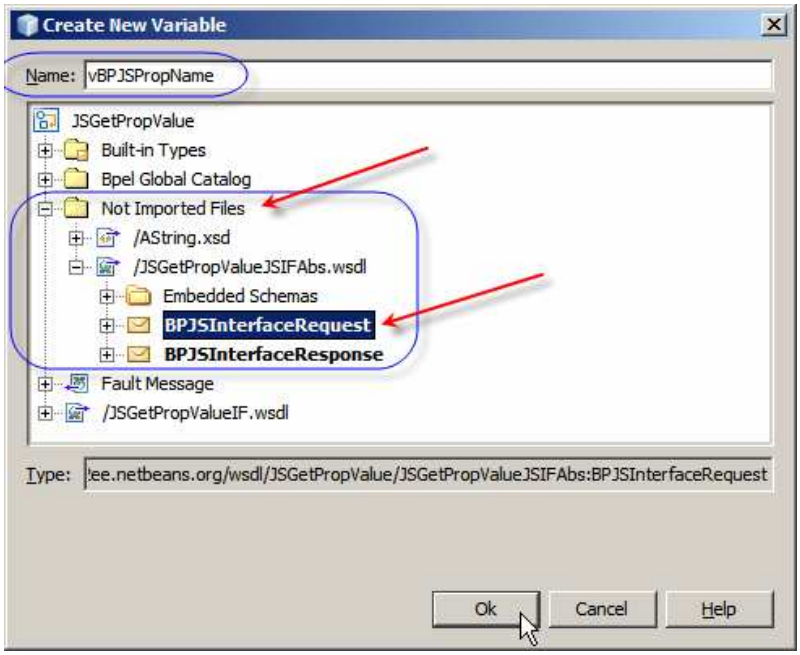
The base process works. We also established the method to test it and to view result of execution.

Stage II

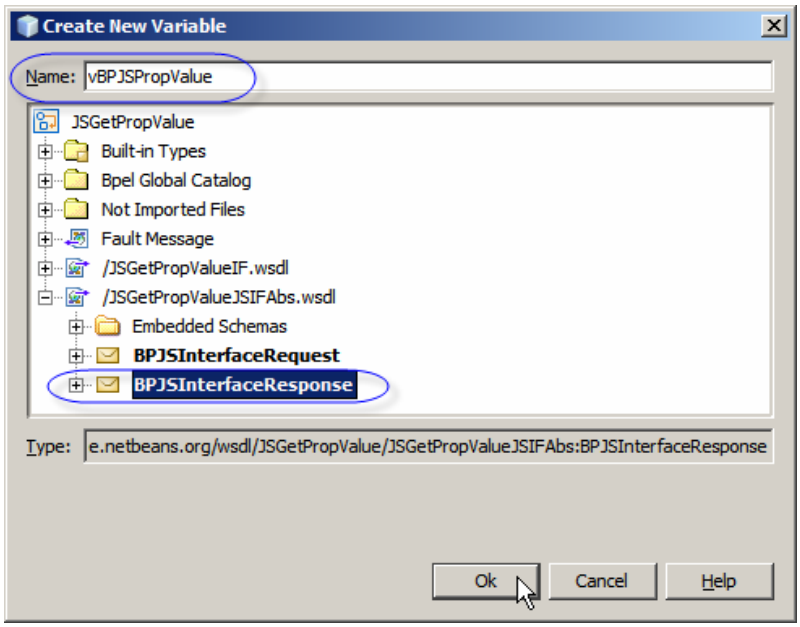
Switch to JSGetPropValue process BPEL editor in Design mode.
Select the outer process scope and click “Add Variable …”



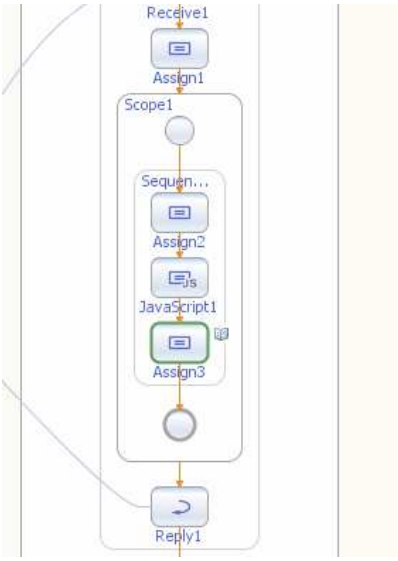
Name the variable vBPJSPropName. Expand “Not Imported Files” through the /JSGetPropValueJSIFAbc.wsdl and choose the BPJSInterfaceRequest.



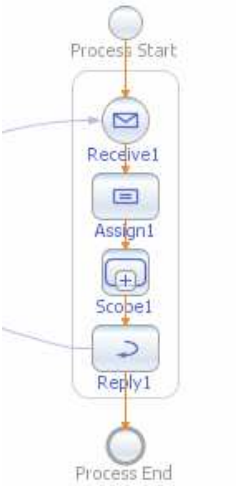
Add another variable. Name it vBPJSPropValue of type BPJSInterfaceResponse.



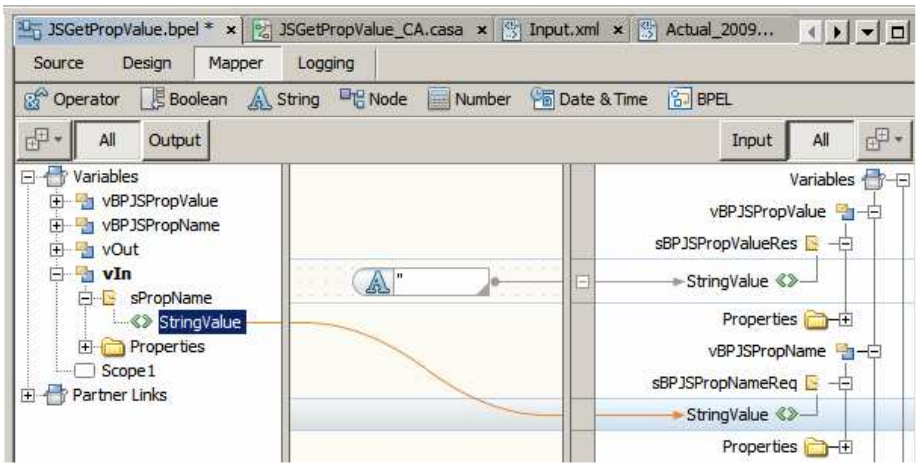
Add a Scope between Assign1 and Reply. To that scope add Assign, JavaScript and Assign, as shown.



Strictly speaking the Scope is not needed. I like to surround embedded JavaScript with a scope for better visibility and so I can collapse the scope and unclutter the process mode, like so:

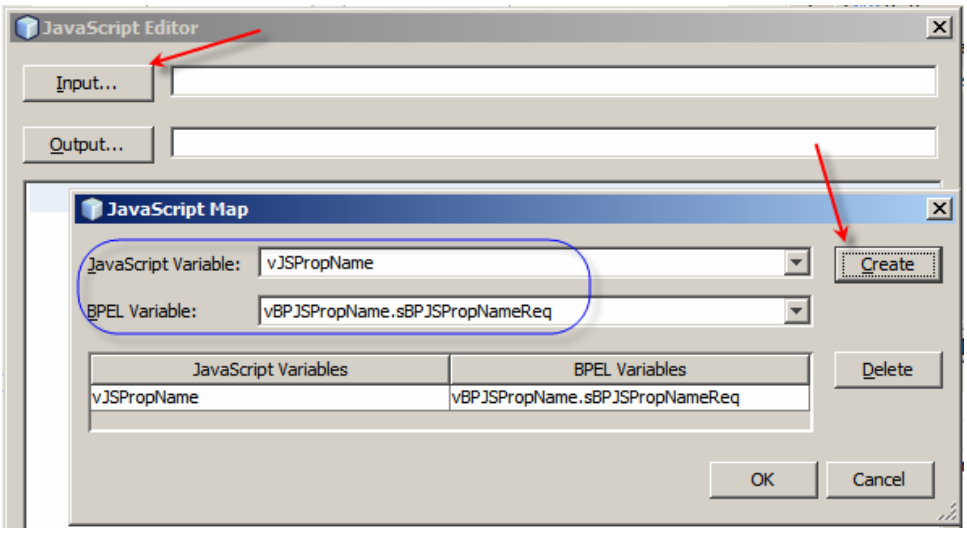


At any rate, select Assign2 and switch to BPEL Mapper. Map vIn to vBPJSPropName and an empty string to vBPJSPropVale as shown.

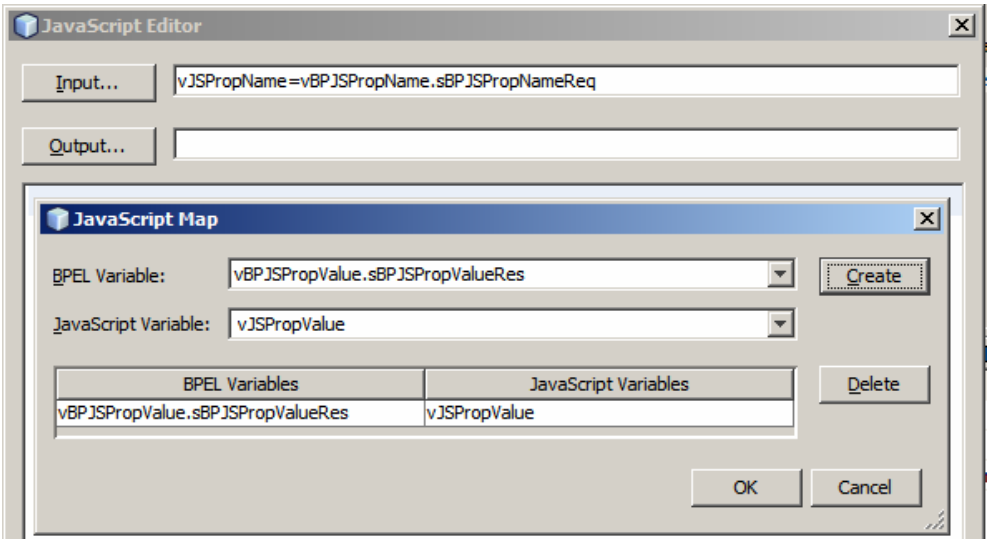


Recall Rule Number 2 – Initialize BPEL variable which is to receive result form JavaScript. This is what we have done above to vBPJSPropValue.

Switch to Design mode. Double-click the JavaScript Activity to open the JavaScript Editor. Click “Input …” button. Provide JavaScript Variable name – vJSPropName. Choose vBOJSPropName.sBPPropNameReq from the dropdown. Click Create button to create the variable association and click OK to complete the dialog.



Repeat the process for the “Output…” side, naming the JavaScript Variable vJSPropValue and choosing vBPJSPropValue.sBPJSPropValueRes from the drop down.



Enter the following JavaScript text into the JavaScript editor window:

```
;
function log(vString) {
    java.lang.System.out.println(vString);
}
log("===>>> Enter Java Script block");
```

```
log("===>>> vJSPropName: " + vJSPropName);

log("===>>> Exit Java Script block");

;
```

In this code block we define a shorthand function, log(), which will write text to the standard output. This is to save us the typing of java.lang.System.out.println every time we need to log something.

We then use the function to log the message that we are entering the JavaScript block, the content of the vJSPropName JavaScript variable and the message that we are exiting the JavaScript code block.

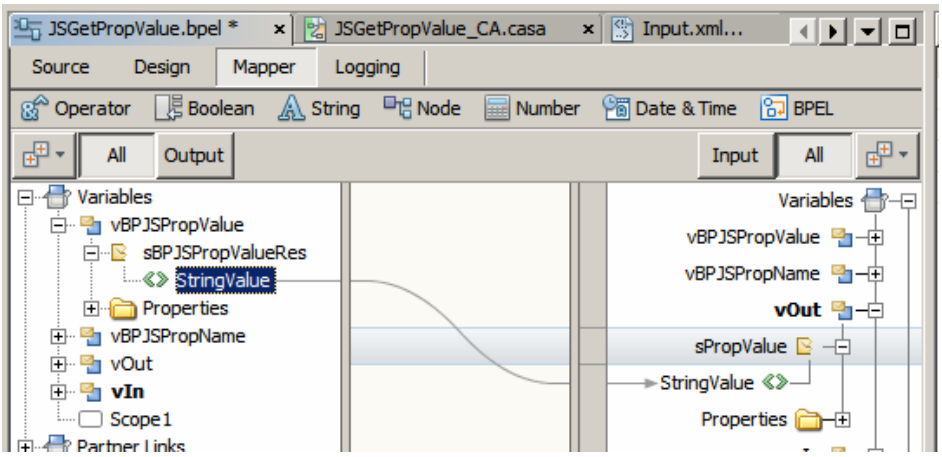
A look at the BPEL source shows something like this:

```
        <scope name="Scope1">
            <sequence name="Sequence1">
                <assign name="Assign2">
                    <copy>
                        <from>$vIn.sPropName/ns0:StringValue</from>
                        <to>$vBPJSPropName.sBPJSPropNameReq/ns0:StringValue</to>
                    </copy>
                    <copy>
                        <from>''</from>
                        <to>$vBPJSPropValue.sBPJSPropValueRes/ns0:StringValue</to>
                    </copy>
                </assign>
                <assign name="JavaScript1">
                    <extensionAssignOperation>
                        <Expression xmlns="http://www.sun.com/wsbpel/2.0/process/executable/SUNExtension/DataHandling" e
function log(vString) {
    java.lang.System.out.println(vSting);
}
log("===>>> Enter Java Script block");

log("===>>> vJSPropName: " + vJSPropName);

log("===>>> Exit Java Script block");

;]]></Expression>
                    </extensionAssignOperation>
                </assign>
                <assign name="Assign3"/>
            </sequence>
        </scope>
```

The “<Expression …” tag hides the details of variable associations. Let’s reformat this to be more readable, like this:

```
                    <extensionAssignOperation>
                        <Expression
                        xmlns="http://www.sun.com/wsbpel/2.0/process/executable/SUNExtension/DataHandling"
                        expressionLanguage="urn:sun:bpel:JavaScript"
                        inputVars="vJSPropName=vBPJSPropName.sBPJSPropNameReq"
                        outputVars="vBPJSPropValue.sBPJSPropValueRes=vJSPropValue">
                            <![CDATA[;
function log(vString) {
    java.lang.System.out.println(vSting);
}
log("===>>> Enter Java Script block");

log("===>>> vJSPropName: " + vJSPropName);

log("===>>> Exit Java Script block");

;]]></Expression>
                    </extensionAssignOperation>
```

Now we need to complete Assign3. Switch to Design mode, select the Assign3 activity and switch to BPEL Mapper mode. Map as shown below.

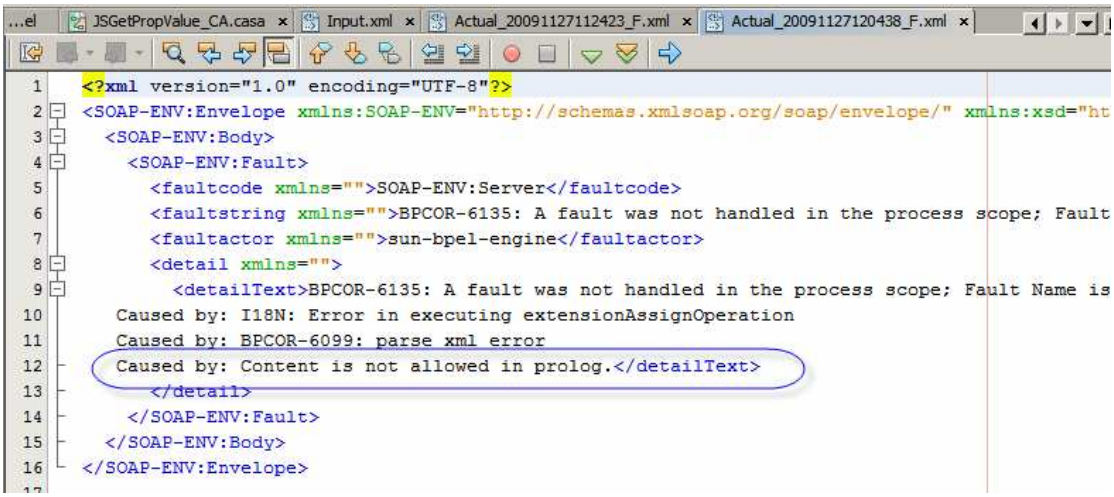


We are using the value of the vBPJSPropValue BPEL variable, which ought to have been set in the JavaScript code but has not been set yet, to populate the web service response message.

Because we are not populating the JavaScript response variable we will get a runtime exception. Before we do we will see the log message in server.log. Let's proceed regardless in order to see what the exception looks like.

Let's build and deploy the composite application and run the test. This time also inspect the server.log to see the logging statements and what they reveal.

Test result shows an exception:



Note the exception: Content is not allowed in prolog. This is not because we are passing invalid message in. This is because we are not populating the JavaScript response variable properly (or not at all in our case so far).

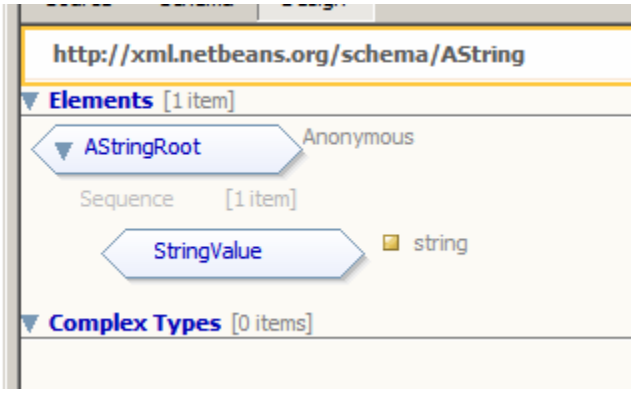
server.log shows the log message from the script block followed by the exception, reinforcing the notion that the issue is in what JavaScript returns to BPEL and not in what BPEL provides to JavaScript.

```
[#|2009-11-27T12:04:35.921+1100|INFO|sun-appserver2.1|javax.enterprise.system.stream.out|_Thr
52f29600b:-7bba;Service Assembly Name=JSGetPropValue_CA;BPEL Process Name=JSGetPropValue;|
===>>> Enter Java Script block|#]

[#|2009-11-27T12:04:35.921+1100|INFO|sun-appserver2.1|javax.enterprise.system.stream.out|_Thr
52f29600b:-7bba;Service Assembly Name=JSGetPropValue_CA;BPEL Process Name=JSGetPropValue;|
===>>> vJSPropName: [AStringRoot: null]|#]

[#|2009-11-27T12:04:35.921+1100|INFO|sun-appserver2.1|javax.enterprise.system.stream.out|_Thr
52f29600b:-7bba;Service Assembly Name=JSGetPropValue_CA;BPEL Process Name=JSGetPropValue;|
===>>> Exit Java Script block|#]

[#|2009-11-27T12:04:35.921+1100|WARNING|sun-appserver2.1|javax.enterprise.system.stream.err|_
:1252f29600b:-7bba;Service Assembly Name=JSGetPropValue_CA;BPEL Process Name=JSGetPropValue;_
in prolog.
|#]

[#|2009-11-27T12:04:35.953+1100|WARNING|sun-appserver2.1|com.sun.jbi.engine.bpel.core.bpel.ut
60.2:-53270e2c:1252f29600b:-7bba;Service Assembly Name=JSGetPropValue_CA;BPEL Process Name=JS
g to createDOM from: org.mozilla.javascript.UniqueTag@11cd14f: NOT_FOUND
 Content is not allowed in prolog.
org.xml.sax.SAXParseException: Content is not allowed in prolog.
        at com.sun.org.apache.xerces.internal.parsers.DOMParser.parse(DOMParser.java:239)
```

Note, too, the content of the sJSPropName is [AStringRoot: null], not the content we assigned to the StringValue node.

This requires a bit of a discussion.

Recall the AString XSD:



The expectation is that an XML instance document which conforms to this schema would look something like:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ns0:AStringRoot
3      xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
4      xmlns:ns0='http://xml.netbeans.org/schema/AString'
5      xsi:schemaLocation='http://xml.netbeans.org/schema/AString AString.xsd'>
6      <ns0:StringValue>I am a value</ns0:StringValue>
7  </ns0:AStringRoot>
```

vJSPropName contains the XML instance document of this structure. We need to parse this XML structure to obtain the value of the StringValue node.

How to manipulate XML in JavaScript is discussed in https://developer.mozilla.org/En/E4X/Processing_XML_with_E4X - Processing XML with JavaScript.

I will not go into in-depth discussion here, merely providing specific statements needed to accomplish the task.

Note that the XML instance document has a `xmlns:ns0` attribute with the value of 'http://xml.netbeans.org/schema/AString'. To work with the nodes in the structure we need to provide a default xml namespace statement, like that shown below. This will make it easier to work with the structure.

```
default xml namespace = 'http://xml.netbeans.org/schema/AString';
```

To eliminate the exception caused by invalid content of the return variable lets explicitly populate it, using one of the two methods supported by E4X.

Let's add the following block of JavaScript, recalling that the return variable must conform to the same structure as input variable:

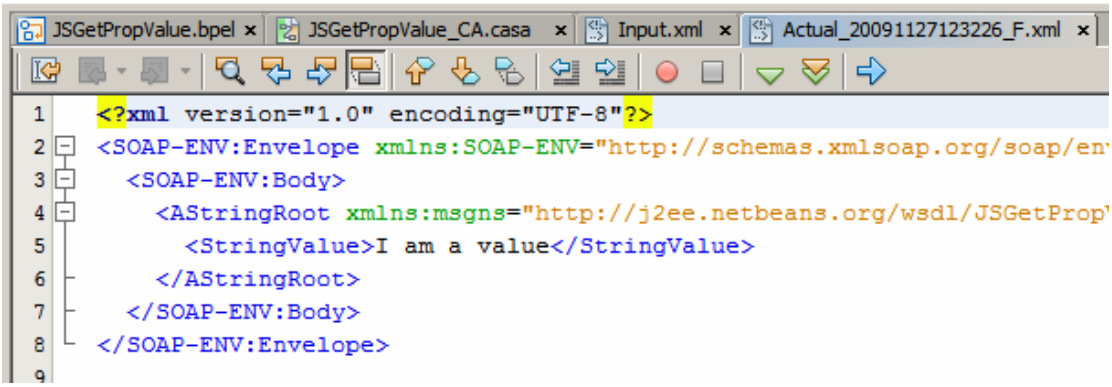
```
vJSPropValue =
<AStringRoot>
    <StringValue>I am a value</StringValue>
</AStringRoot>;
```

Because we have the default xml namespace statement in the script the default namespace will be used for this structure.

Our script now looks like this:

```
;
function log(vString) {
    java.lang.System.out.println(vString);
}
log("===>>> Enter Java Script block");

default xml namespace = 'http://xml.netbeans.org/schema/AString';

log("===>>> vJSPropName: " + vJSPropName);

vJSPropValue =
<AStringRoot>
    <StringValue>I am a value</StringValue>
</AStringRoot>;

log("===>>> Exit Java Script block");

;
```

Build, deploy and test the composite application. Review the test result and see:



We no longer get an exception, meaning that the return structure is properly populated. We also see that the namespace has been added.

Let's now turn our attention to the input message. The JavaScript variable vJSPropName points to the AStringRoot. To get the value of the StringValue node we need statements like:

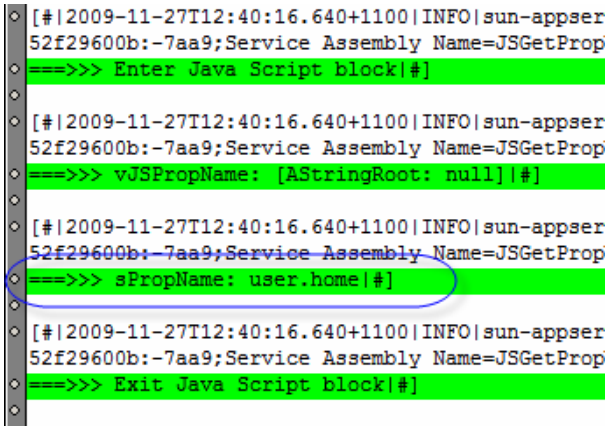
```
var vJSPropNameXML = new XML(vJSPropName);
var sPropName = vJSPropNameXML.StringValue;
log("===>>> sPropName: " + sPropName);
```

The script thus far looks like:

```
;
function log(vString) {
    java.lang.System.out.println(vString);
}
log("===>>> Enter Java Script block");

default xml namespace = 'http://xml.netbeans.org/schema/AString';

log("===>>> vJSPropName: " + vJSPropName);

var vJSPropNameXML = new XML(vJSPropName);
var sPropName = vJSPropNameXML.StringValue;
log("===>>> sPropName: " + sPropName);


vJSPropValue =
<AStringRoot>
    <StringValue>I am a value</StringValue>
</AStringRoot>;

log("===>>> Exit Java Script block");

;
```

Build, deploy and test the composite application.

Now server.log says:



We are correctly referencing the StringValue node in the structure and getting hold of the value, that is the name of the property passed from BPEL. We are now in a position to use this property name to get the property value and to set it in the result.

Add the following statements:

```
var sPropValue = java.lang.System.getProperty(sPropName);
```

and

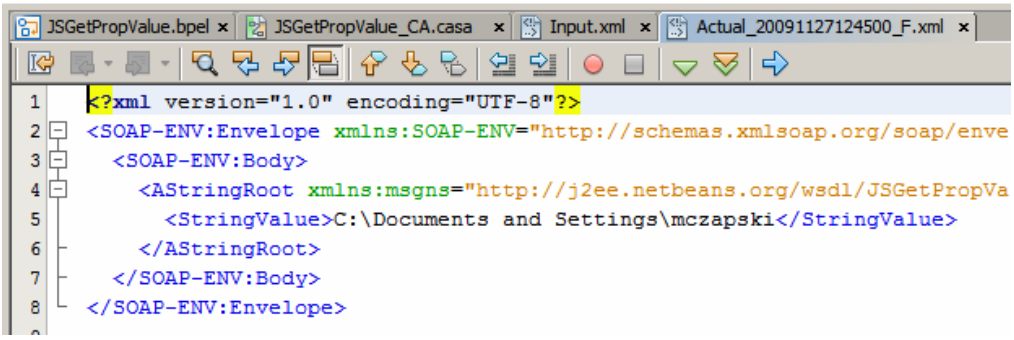
```
vJSPropValue.StringValue = sPropValue;
```

The complete script looks like this:

```
;
function log(vString) {
    java.lang.System.out.println(vString);
}
log("===>>> Enter Java Script block");

default xml namespace = 'http://xml.netbeans.org/schema/AString';

log("===>>> vJSPropName: " + vJSPropName);

var vJSPropNameXML = new XML(vJSPropName);
var sPropName = vJSPropNameXML.StringValue;
log("===>>> sPropName: " + sPropName);

var sPropValue = java.lang.System.getProperty(sPropName);

vJSPropValue =
<AStringRoot>
    <StringValue>I am a value</StringValue>
</AStringRoot>;

vJSPropValue.StringValue = sPropValue;


log("===>>> Exit Java Script block");

;
```

Build, deploy and test the composite application.

Inspecting the test result I see:

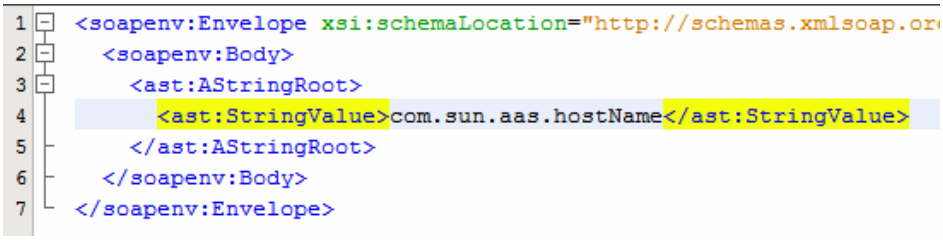


The embedded JavaScript code block returns the value of the property whose name is given to it as input.

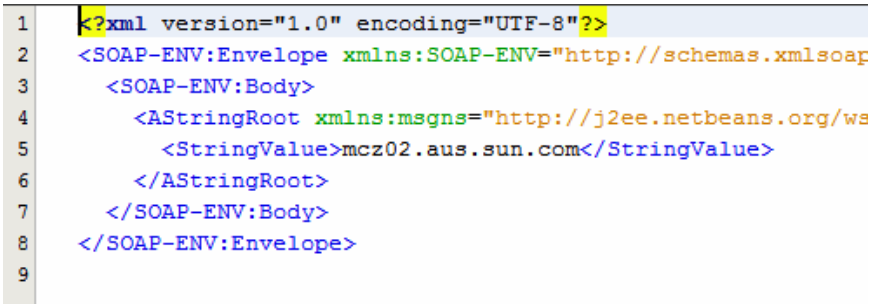
Now that the script is demonstrably working we can get rid of the logging statements.

```
;
default xml namespace = 'http://xml.netbeans.org/schema/AString';
var vJSPropNameXML = new XML(vJSPropName);
var sPropName = vJSPropNameXML.StringValue;
var sPropValue = java.lang.System.getProperty(sPropName);
vJSPropValue =
<AStringRoot>
    <StringValue/>
</AStringRoot>;
vJSPropValue.StringValue = sPropValue;
;
```

Let's modify the test input message and set the following string as property name:



Let's run a test and see the result.



With a bit of JavaScript and a bit of Java we can determine the host name of the host on which the BPEL SU executes. In a clustered environment this may be useful to know.

Example 2: Date Format Conversion

Let's now look again at the date conversion code. Here it is as it was when we run it under the Rhino Shell and the Rhino Debugger.

```
// begin JavaScript code
//
var vDateTimeIn = arguments[0];
var vDateTimeInFormat = arguments[1];
var vDateTimeOutFormat = arguments[2];

function cvtDateTime(vDateTimeIn, vDateTimeInFormat, vDateTimeOutFormat) {
//     var vDateTimeIn = "10/12/1956";
//     var vDateTimeInFormat = "dd/MM/yyyy";
//     var vDateTimeOutFormat = "dd-MM-yyyy'T'hh:mm:ss";
  fmtIn = new java.text.SimpleDateFormat(vDateTimeInFormat);
  fmtOut = new java.text.SimpleDateFormat(vDateTimeOutFormat);
  var dt = fmtIn.parse(vDateTimeIn);
  return fmtOut.format(dt);
}

var sDateTimeOut = cvtDateTime
        (vDateTimeIn
        ,vDateTimeInFormat
        ,vDateTimeOutFormat);

java.lang.System.out.println("Converted Date: " + sDateTimeOut);

//
// end JavaScript code
```

A brief analysis will reveal that we need 3 pieces of information as input to this script, `DateTimeIn, DateTimeInFormat` and `DateTimeOutFormat`. The function returns a single output string, `DateTimeOut`.

From Rule Number 1 we recall the need to have a messageType for input and output, via an Abstract WSDL, with two XML Schema-compliant messages, one for input and one for output.

Let's create a BPLE Module project, JSCvtDateTime.

Let's create a XML Schema document, CvtDateTimeSchema, which satisfies the requirements of the interface, as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://xml.netbeans.org/schema/CvtDateTimeSchema"
    xmlns:tns="http://xml.netbeans.org/schema/CvtDateTimeSchema"
    elementFormDefault="qualified">
    <xsd:element name="CvtDTReq">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="DateTimeIn" type="xsd:string"/>
                <xsd:element name="DateTimeFmtIn" type="xsd:string"/>
                <xsd:element name="DateTimeFmtOut" type="xsd:string"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="CvtDTRes">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="DateTimeOut" type="xsd:string"/>
```

```
            </xsd:sequence>
         </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

The CvtDTReq structure will be used to pass input data. The CvtDTRes structure will be used to receive the result.

Here is a request instance document:

```
<CvtDTReq xmlns="http://xml.netbeans.org/schema/CvtDateTimeSchema">
       <DateTimeIn>10-12-1956</DateTimeIn>
       <DateTimeFmtIn>dd-MM-yyyy</DateTimeFmtIn>
       <DateTimeFmtOut>MMMM dd, yyyy</DateTimeFmtOut>
</CvtDTReq>
```

Here is a response instance document:

```
<CvtDTRes xmlns="http://xml.netbeans.org/schema/CvtDateTimeSchema">
       <DateTimeOut>December 10, 1956</DateTimeOut >
</CvtDTRes>
```

Let's create a concrete WSDL, which we will use as an interface to expose the process as a web service, to facilitate testing. Let's name this WSDL **JSVctDateTimeIF** and configure it as follows.

WSDL Type: **Concrete**
Binding: **SOAP**
Type: **Document Literal**
Operation Name: **cvtDT**
Input:
Message Part Name: **sCvtDTReq**
Element Or Type: **CvtDTReq**
Output:
Message Part Name: **sCvtDTRes**
Element Or Type: **CvtDTRes**

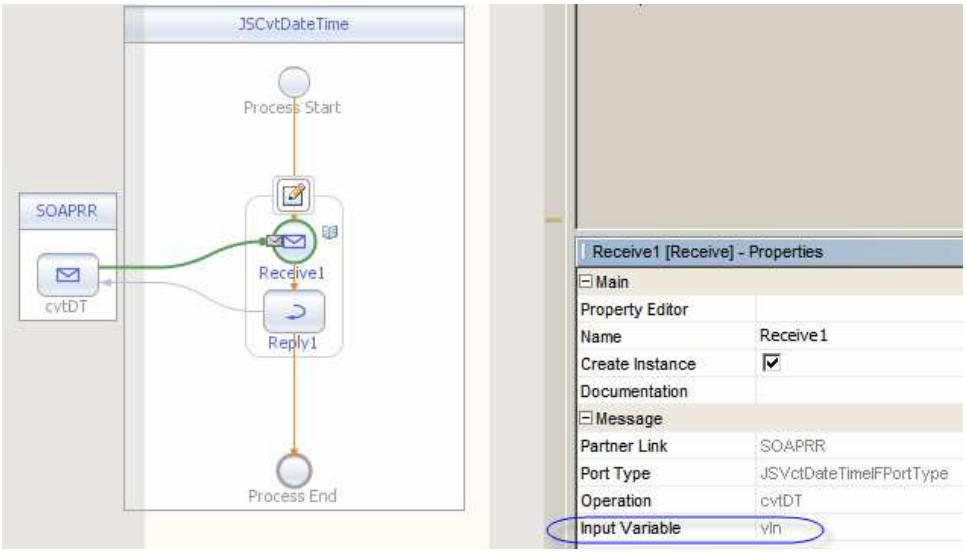
Let's create an Abstract WSDL, **JSCvtDateTimeIFAbs**, as follows:

WSDL Type: **Abstract**
Operation Name: **cvtDT**
Input:
Message Part Name: **sCvtDTReq**
Element Or Type: **CvtDTReq**
Output:
Message Part Name: **sCvtDTRes**
Element Or Type: **CvtDTRes**

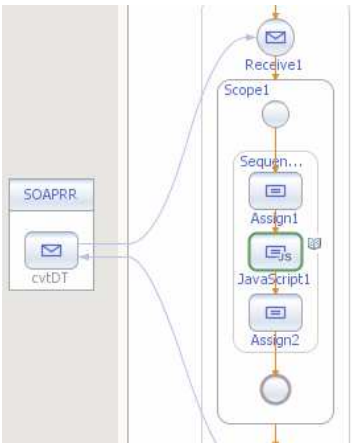
Notice that we reused the message structures. This is fine in the exercise. In a real project we are much more likely to have a different XML schema for the service interface and have a schema needed for the JavaScript code as a separate special-purpose schema.

Let's open the JSCvtDateTime BPEL and add the JSVctDateTimeIF WSDL to the left hand side swim line. Add Receive and Reply activities, connect to the partner link and

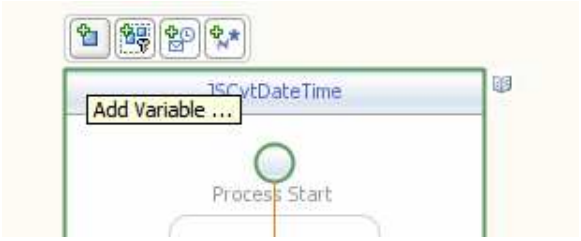
configure input (vIn) and output (vOut) variables. This is the same as in the previous exercise.



Add a scope between Receive1 and Reply1. Add Assign, JavaScript and Assign into the scope. As before, the Scope is not really needed. I use it for "grouping" purposes.



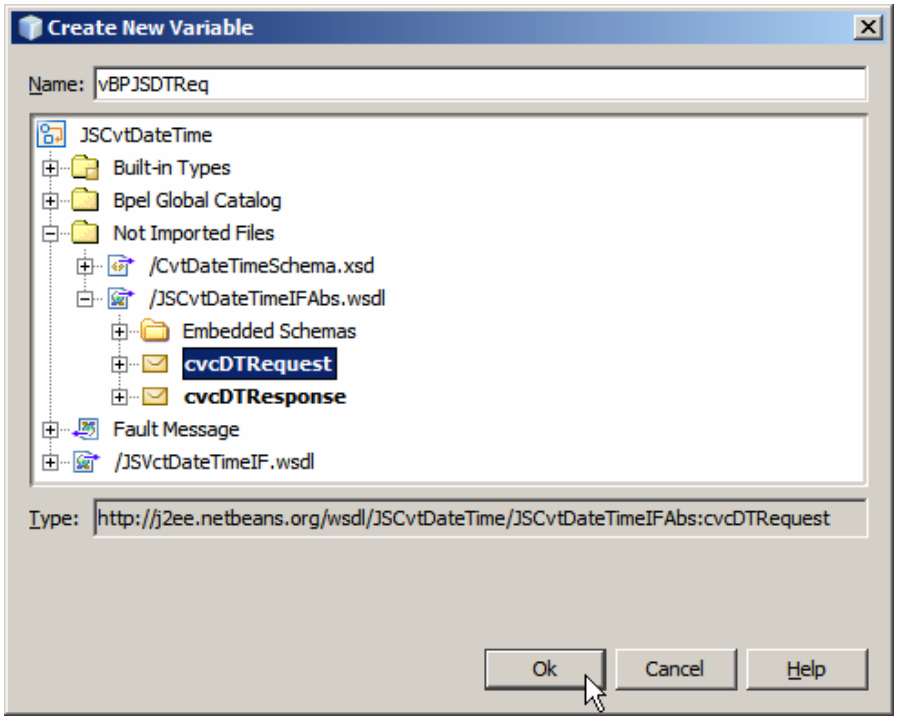
Select the process scope and click Add Variable …



Variable:

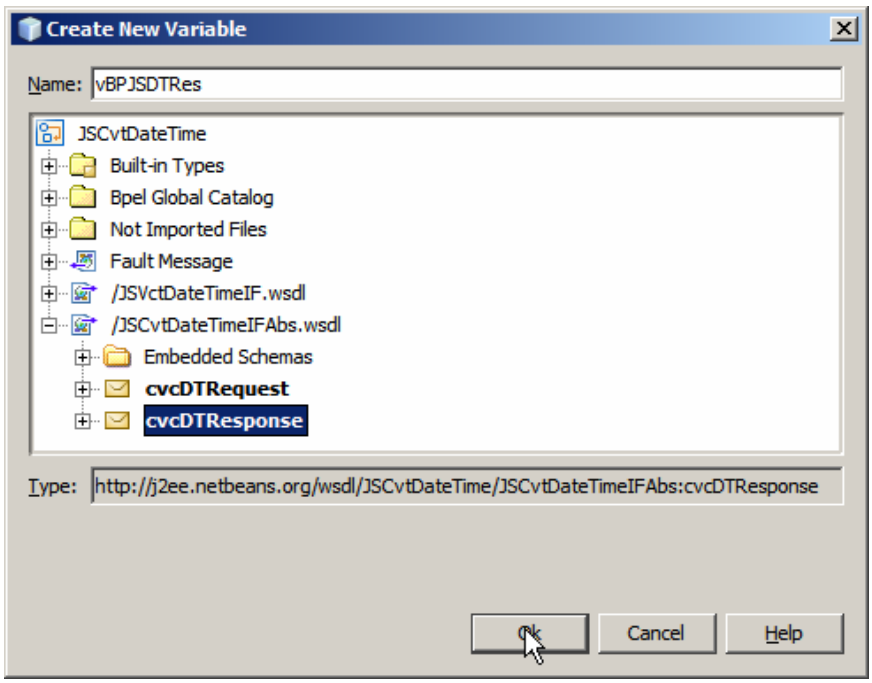
Name: vBPJSDTReq

Type: Not Imported Files → /JSCvtDateTimeIFAbs → cvcDTRequest

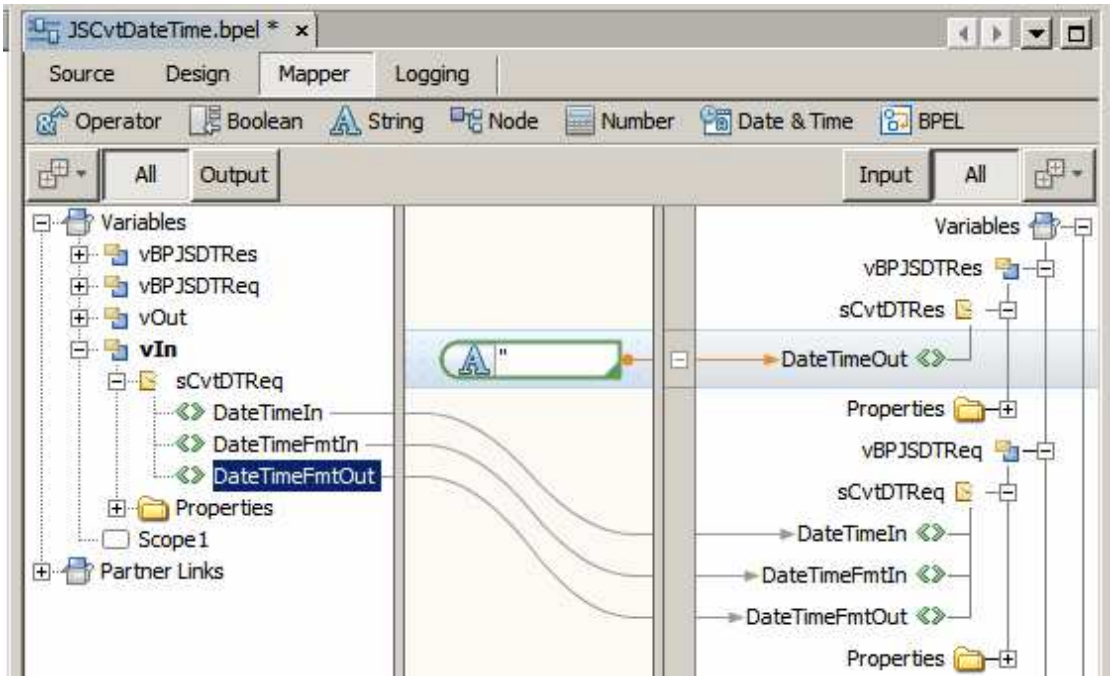


Add another variable:
Variable:

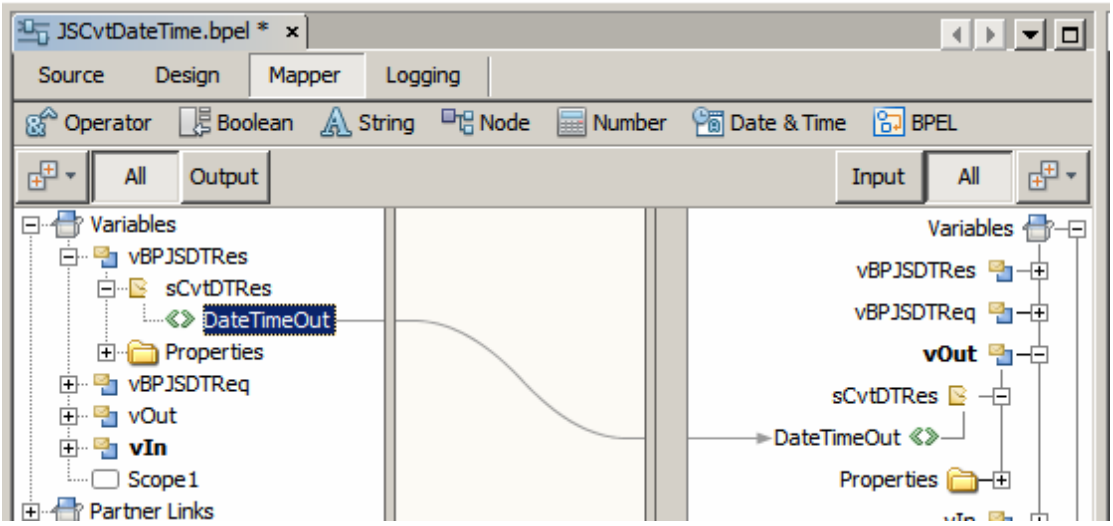
Name: vBPJSDTRes
Type: /JSCvtDateTimeIFAbs → cvcDTResponse



Map Assign1 as follows, remembering Rule Number 2 – initialize request and response variable before use:

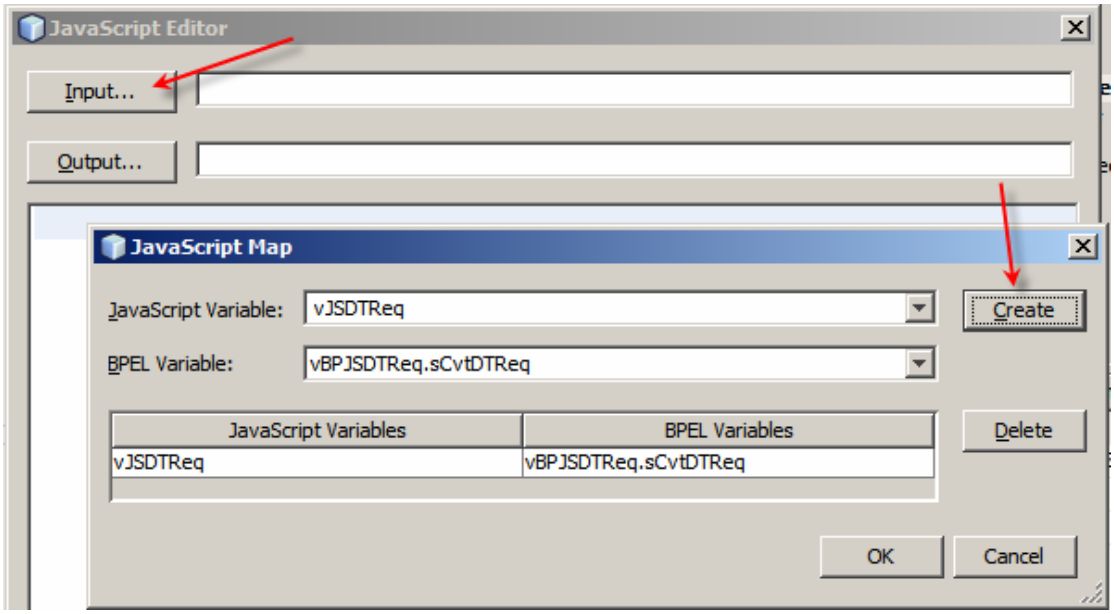


Map Assign2 as follows:



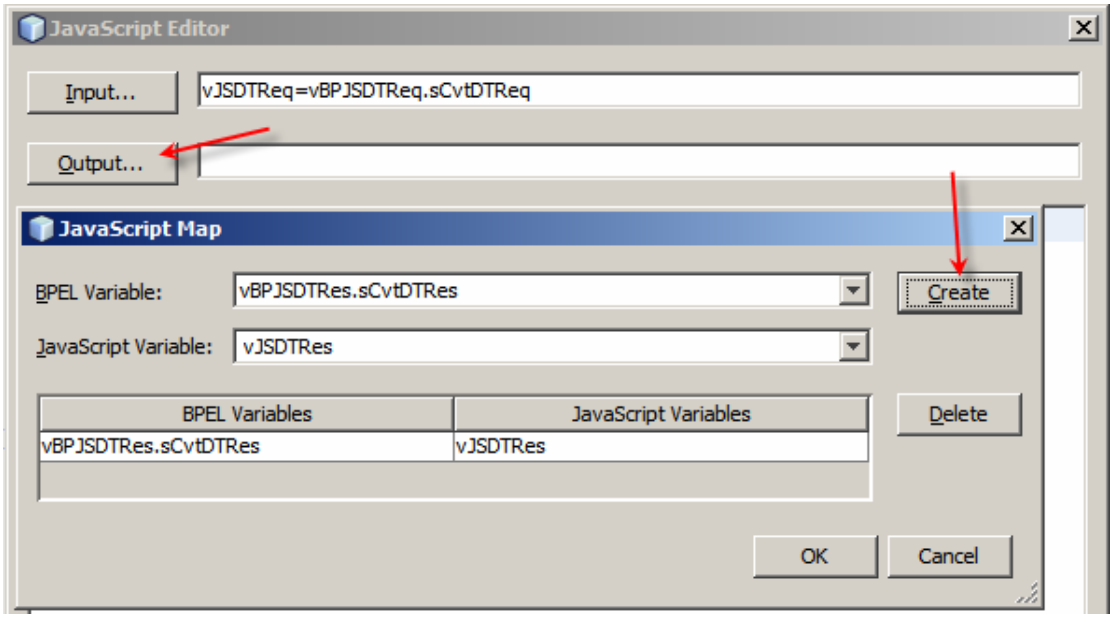
Switch to Design view, double-click the JavaScript activity to open the JavaScript editor and click the Input… button.

Name JavaScript Variable vJSDTReq, choose the vBPJSDTReq.sCvtDTReq BPEL variable, then click Create and OK to make the association.



Click the Output… button.

Choose the vBPJSDTRes.sCvtDTRes BPEL variable, name JavaScript Variable vJSDTRes, click Create and OK to make the association.



The XML Instance document corresponding to the Request in CvtDateTimeSchema:

```
<CvtDTReq xmlns="http://xml.netbeans.org/schema/CvtDateTimeSchema">
       <DateTimeIn>10-12-1956</DateTimeIn>
       <DateTimeFmtIn>dd-MM-yyyy</DateTimeFmtIn>
       <DateTimeFmtOut>MMMM dd, yyyy</DateTimeFmtOut>
</CvtDTReq>
```

The Response would look like this:

```
<CvtDTRes xmlns="http://xml.netbeans.org/schema/CvtDateTimeSchema">
       <DateTimeOut>11/11/1111</DateTimeOut>
```

```
</CvtDTRes>
```

We have:

```
BPEL
      vBPJSDTReq.sCvtDTReq → vJSDTReq
            JavaScript code
      vJSDTRes → vBPJSDTRes.sCvtDTRes
BPEL
```

To access the individual node values in JavaScript we would use code like:

```
default xml namespace = "http://xml.netbeans.org/schema/CvtDateTimeSchema";
var vJSDTReqXML = new XML(vJSDTReq);
var sDateTimeIn = vJSDTReqXML.DateTimeIn;
var sDateTimeFmtIn = vJSDTReqXML.DateTimeFmtIn;
var sDateTimeFmtOut = vJSDTReqXML.DateTimeFmtOut;
```

To construct a skeleton response we would use code like:

```
vJSDTRes = <CvtDTRes>
    <DateTimeOut/>
</CvtDTRes>;
```

The entire JavaScript block, derived from the CvtDateTime function we developed and exercised in Section “Developing JavaScript with embedded Java”, and modified to use in BPEL would look like this:

```
// begin JavaScript code
//
function log(vStr) {
    java.lang.System.out.println(vStr);
}

log("===>>> Entering JavaScript");
default xml namespace = "http://xml.netbeans.org/schema/CvtDateTimeSchema";

var vJSDTReqXML = new XML(vJSDTReq);

var sDateTimeIn = vJSDTReqXML.DateTimeIn;
var sDateTimeFmtIn = vJSDTReqXML.DateTimeFmtIn;
var sDateTimeFmtOut = vJSDTReqXML.DateTimeFmtOut;

function cvtDateTime(vDateTimeIn, vDateTimeInFormat, vDateTimeOutFormat) {
  fmtIn = new java.text.SimpleDateFormat(vDateTimeInFormat);
  fmtOut = new java.text.SimpleDateFormat(vDateTimeOutFormat);
  var dt = fmtIn.parse(vDateTimeIn);
  return fmtOut.format(dt);
}

var sDateTimeOut = cvtDateTime
        (sDateTimeIn
        ,sDateTimeFmtIn
        ,sDateTimeFmtOut);

vJSDTRes = <CvtDTRes>
    <DateTimeOut/>
</CvtDTRes>;

vJSDTRes.DateTimeOut = sDateTimeOut;

log("===>>> Exiting JavaScript");
```

```
//
// end JavaScript code
```

Add this script to the JavaScript editor window and build the project.

Create a new composite application, JSCvtDateTime_CA, drag the BPEL module onto the CASA canvas, build and deploy.

Create a New test Case in the JSCvtDateTime_CA project, populate the Input message with appropriate values and run the test.

```
<soapenv:Envelope xsi:schemaLocation="http://schemas.xmlsoap.o
  <soapenv:Body>
    <cvt:CvtDTReq>
      <cvt:DateTimeIn>10-12-1956</cvt:DateTimeIn>
      <cvt:DateTimeFmtIn>dd-MM-yyyy</cvt:DateTimeFmtIn>
      <cvt:DateTimeFmtOut>MMMM dd, yyyy</cvt:DateTimeFmtOut>
    </cvt:CvtDTReq>
  </soapenv:Body>
</soapenv:Envelope>
```

The test response:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soa
  <SOAP-ENV:Body>
    <CvtDTRes xmlns:msgns="http://j2ee.netbeans.org/wsdl/JSCvtDat
      <DateTimeOut>December 10, 1956</DateTimeOut>
    </CvtDTRes>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

When all is well, the code works each time and every time. When it does not work it may be difficult to figure out what the issue is.

Let’s say we have this script, which looks right but does not work:

```
// begin JavaScript code
//
function log(vStr) {
    java.lang.System.out.println(vStr);
}

log("===>>> Entering JavaScript");
default xml namespace = "http://xml.netbeans.org/schema/CvtDateTimeSchema";

var vJSDTReqXML = new XML(vJSDTReq);

var sDateTimeIn = vJSDTReqXML.DateTimeIn;
var sDateTimeFmtIn = vJSDTReqXML.DateTimeFmtIn;
var sDateTimeFmtOut = vJSDTReq.DateTimeFmtOut;

function cvtDateTime(vDateTimeIn, vDateTimeInFormat, vDateTimeOutFormat) {
  fmtIn = new java.text.SimpleDateFormat(vDateTimeInFormat);
  fmtOut = new java.text.SimpleDateFormat(vDateTimeOutFormat);
  var dt = fmtIn.parse(vDateTimeIn);
  return fmtOut.format(dt);
}

var sDateTimeOut = cvtDateTime
```

```
        (sDateTimeIn
        ,sDateTimeFmtIn
        ,sDateTimeFmtOut);

vJSDTRes = <CvtDTRes>
    <DateTimeOut/>
</CvtDTRes>;

vJSDTRes.DateTimeOut = sDateTimeOut;

log("===>>> Exiting JavaScript");
//
// end JavaScript code
```

The test results say something like:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://schemas.xmlsoap.org/soap/envelope/
http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <faultcode xmlns="">SOAP-ENV:Server</faultcode>
      <faultstring xmlns="">BPCOR-6135: A fault was not handled in the process scope;
Fault Name is
{http://www.sun.com/wsbpel/2.0/process/executable/SUNExtension/ErrorHandling}systemFau
lt; Fault Data is &amp;lt;?xml version=&amp;quot;1.0&amp;quot; encoding=&amp;quot;UTF-
8&amp;quot;?&amp;gt;&amp;lt;jbi:message
xmlns:sxeh=&amp;quot;http://www.sun.com/wsbpel/2.0/process/executable/SUNExtension/Err
orHandling&amp;quot; type=&amp;quot;sxeh:faultMessage&amp;quot;
version=&amp;quot;1.0&amp;quot;
xmlns:jbi=&amp;quot;http://java.sun.com/xml/ns/jbi/wsdl-11-
wrapper&amp;quot;&amp;gt;&amp;lt;jbi:part&amp;gt;Wrapped
java.lang.IllegalArgumentException: Illegal pattern character &amp;apos;u&amp;apos;
(extensionAssignOperation.Expression#19)&amp;lt;/jbi:part&amp;gt;&amp;lt;/jbi:message&
amp;gt;. Sending errors for the pending requests in the process scope before
terminating the process instance</faultstring>
      <faultactor xmlns="">sun-bpel-engine</faultactor>
      <detail xmlns="">
        <detailText>BPCOR-6135: A fault was not handled in the process scope; Fault
Name is
{http://www.sun.com/wsbpel/2.0/process/executable/SUNExtension/ErrorHandling}systemFau
lt; Fault Data is &amp;lt;?xml version=&amp;quot;1.0&amp;quot; encoding=&amp;quot;UTF-
8&amp;quot;?&amp;gt;&amp;lt;jbi:message
xmlns:sxeh=&amp;quot;http://www.sun.com/wsbpel/2.0/process/executable/SUNExtension/Err
orHandling&amp;quot; type=&amp;quot;sxeh:faultMessage&amp;quot;
version=&amp;quot;1.0&amp;quot;
xmlns:jbi=&amp;quot;http://java.sun.com/xml/ns/jbi/wsdl-11-
wrapper&amp;quot;&amp;gt;&amp;lt;jbi:part&amp;gt;Wrapped
java.lang.IllegalArgumentException: Illegal pattern character &amp;apos;u&amp;apos;
(extensionAssignOperation.Expression#19)&amp;lt;/jbi:part&amp;gt;&amp;lt;/jbi:message&
amp;gt;. Sending errors for the pending requests in the process scope before
terminating the process instance
   Caused by: I18N: Error in executing extensionAssignOperation
   Caused by: Wrapped java.lang.IllegalArgumentException: Illegal pattern character
&amp;apos;u&amp;apos; (extensionAssignOperation.Expression#19)
   Caused by: Illegal pattern character
&amp;apos;u&amp;apos;</detailText>
      </detail>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

One wonders what that might possibly mean and where the issue might lie.

One suspects the issue is somewhere in JavaScript. How does one get to load this code into a JavaScript Debugger so one can debug it?

We know that the input to this script is a vJSDTReq and we know what it is supposed to look like.

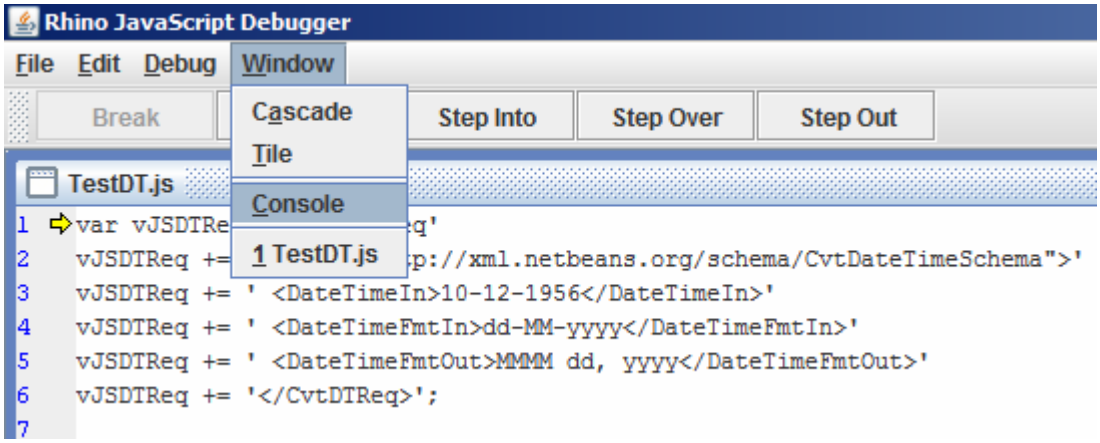
Let's paste the entire script into a text file, say TestDT.js, and add statements which initialize vJSDTReq to an appropriate value and display the result.

```
var vJSDTReq = <CvtDTReq
        xmlns="http://xml.netbeans.org/schema/CvtDateTimeSchema">
        <DateTimeIn>10-12-1956</DateTimeIn>
        <DateTimeFmtIn>dd-MM-yyyy</DateTimeFmtIn>
        <DateTimeFmtOut>MMMM dd, yyyy</DateTimeFmtOut>
</CvtDTReq>;

// begin JavaScript code
//
function log(vStr) {
    java.lang.System.out.println(vStr);
}

log("===>>> Entering JavaScript");
default xml namespace = "http://xml.netbeans.org/schema/CvtDateTimeSchema";

var vJSDTReqXML = new XML(vJSDTReq);

var sDateTimeIn = vJSDTReqXML.DateTimeIn;
var sDateTimeFmtIn = vJSDTReqXML.DateTimeFmtIn;
var sDateTimeFmtOut = vJSDTReq.DateTimeFmtOut;

function cvtDateTime(vDateTimeIn, vDateTimeInFormat, vDateTimeOutFormat) {
  fmtIn = new java.text.SimpleDateFormat(vDateTimeInFormat);
  fmtOut = new java.text.SimpleDateFormat(vDateTimeOutFormat);
  var dt = fmtIn.parse(vDateTimeIn);
  return fmtOut.format(dt);
}

var sDateTimeOut = cvtDateTime
        (sDateTimeIn
        ,sDateTimeFmtIn
        ,sDateTimeFmtOut);

vJSDTRes = <CvtDTRes>
    <DateTimeOut/>
</CvtDTRes>;

vJSDTRes.DateTimeOut = sDateTimeOut;

log("===>>> Exiting JavaScript");
//
// end JavaScript code


log("===>>> vJSDTRes: " + vJSDTRes.toString());
```

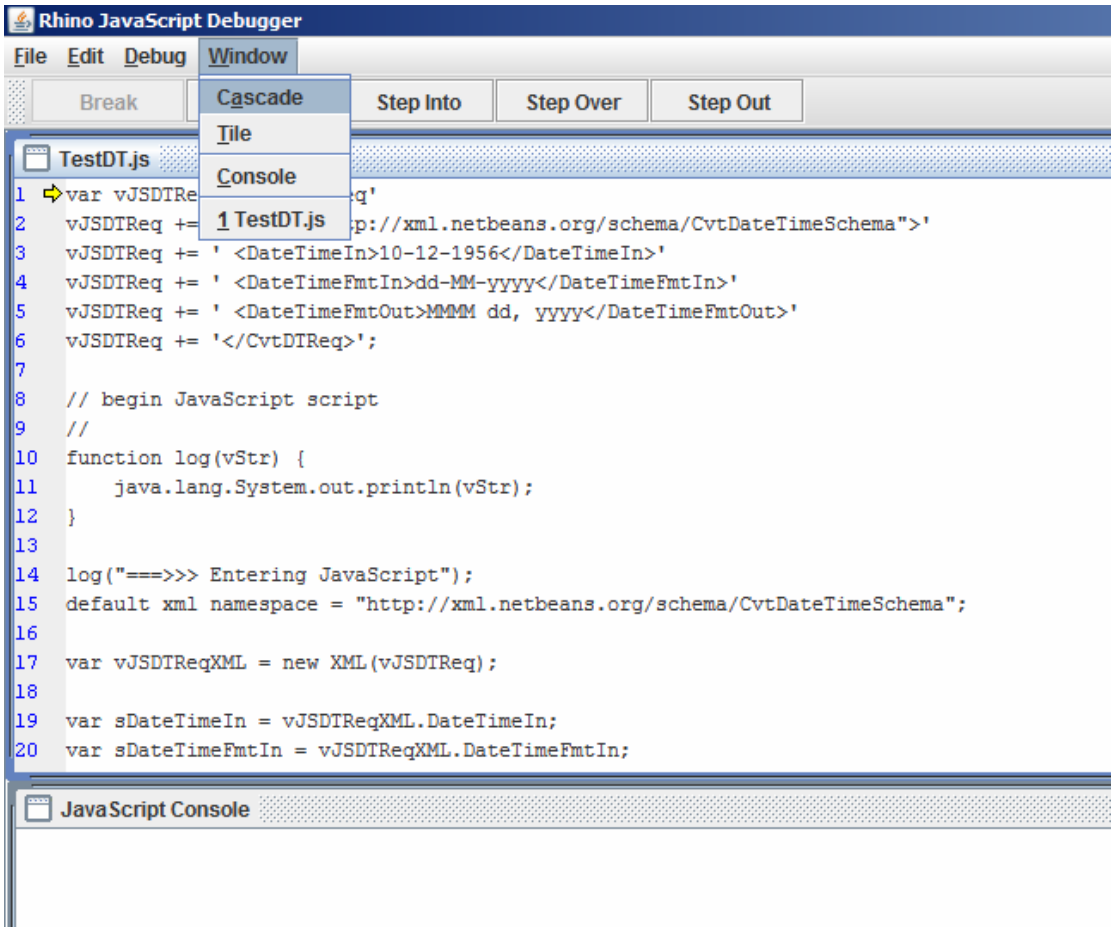
Let's now submit it, to the Rhino JavaScript Debugger, with a command line like:

```
C:\>%JAVA_HOME%\bin\java.exe -cp c:\tools\rhino1_7R2\js.jar
org.mozilla.javascript.tools.debugger.Main TestDT.js
```

Open the Console Window, so any exceptions can be seen:



Cascade and rearrange the windows so you can see both the code being executed and the standard output and standard error in the console window.



Execute the code a line at a time until something happens.

For me the exception points out to line 25 in the script. This line manipulates vDateTimeOutFormat.

```
22
23  function cvtDateTime(vDateTimeIn, vDateTimeInFormat, vDateTimeOutFormat) {
24    fmtIn = new java.text.SimpleDateFormat(vDateTimeInFormat);
25    fmtOut = new java.text.SimpleDateFormat(vDateTimeOutFormat);
26    var dt = fmtIn.parse(vDateTimeIn);
27    return fmtOut.format(dt);
28  }
29
30  var sDateTimeOut = cvtDateTime
31          (sDateTimeIn
32           sDateTimeFmtIn
```

```
JavaScript Console
===>>> Entering JavaScript
org.mozilla.javascript.WrappedException: Wrapped java.lang.IllegalArgumentException: Illegal pattern character 'u' (TestDT.js#25)
        at org.mozilla.javascript.Context.throwAsScriptRuntimeEx(Context.java:1773)
        at org.mozilla.javascript.MemberBox.newInstance(MemberBox.java:202)
        at org.mozilla.javascript.NativeJavaClass.constructSpecific(NativeJavaClass.java:281)
        at org.mozilla.javascript.NativeJavaClass.construct(NativeJavaClass.java:200)
        at org.mozilla.javascript.Interpreter.interpretLoop(Interpreter.java:3377)
```

vDateTimeOutFormat variable, which is an argument to the cvtDateTime function, is initialized at line 33 from the variable sDateTimeFmtOut.

```
30  var sDateTimeOut = cvtDateTime
31          (sDateTimeIn
32          ,sDateTimeFmtIn
33          ,sDateTimeFmtOut);
34
```

Variable sDateTimeFmtOut is set at line 21:

```
20  var sDateTimeFmtIn = vJSDTReqXML.DateTimeFmtIn;
21  var sDateTimeFmtOut = vJSDTReq.DateTimeFmtOut;
22
```

Looking into Context at the value of this variable reveals that it is undefined.

```
        at org.mozilla.javascript.InterpretedFunction.call(InterpretedFunction.j
        at org.mozilla.javascript.ContextFactory.doTopCall(ContextFactory.java:3
```

Context: "TestDT.js", line 30

Name	Value
readUrl	function readUrl() { [native code, arity=1] }
ReferenceError	function ReferenceError() { [native code for ReferenceError, arity=1] }
RegExp	function RegExp() { [native code, arity=0] }
runCommand	function runCommand() { [native code, arity=1] }
Script	TypeError: Method "toString" called on incompatible object.
sDateTimeFmtIn	dd-MM-yyyy
sDateTimeFmtOut	undefined
sDateTimeIn	10-12-1956
sDateTimeOut	undefined
seal	function seal() { [native code, arity=1] }

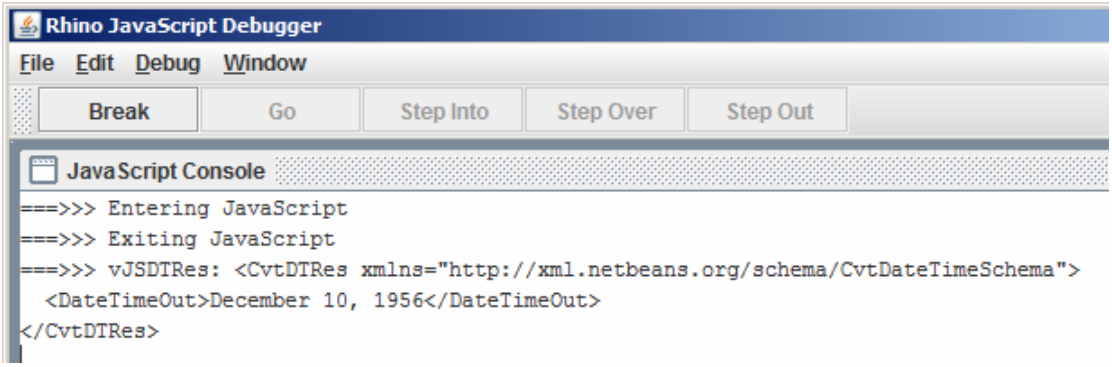
This tells me that there is something wrong with the assignment:

```
20  var sDateTimeFmtIn = vJSDTReqXML.DateTimeFmtIn;
21  var sDateTimeFmtOut = vJSDTReq.DateTimeFmtOut;
22
```

Looking closely tells me that I am using the wrong object to get the vale for this variable. I should have used vJSDTReqXML. Let's change the code and execute it again.

This time sDateTimeOutFmt gets set correctly and the script completes normally.

The Debugger Console window shows:



Now we have a reasonable degree of confidence that this script, minus the variable initialization and termination code we added to be able to debug the script, will work in BPEL.

Summary

In this Note we explored the BPEL SE capability to execute JavaScript code inline. In passing we also explored the ability of JavaScript to execute Java statements and through these means to extend BPEL 2.0 with arbitrarily sophisticated functionality without having to resort to invoking web services or POJOs.

We introduced the 2 Rules which must be followed, and 1 Rule which should be followed, for successful BPEL and JavaScript integration. We developed two complete examples of embedded JavaScript code that provided reasonably useful functionality not natively available through BPEL. While the two examples were fairly trivial it is clear that more sophisticated functionality can be added following the method introduced in this Note.

Next release of GlassFish ESB, release 2.2, will add a number of improvements, including the ability to invoke a static Java class as a function directly in the BPEL Mapper.