Web-Enable Your SAP Applications with the Power of JavaScript

Peter Januschke and Holger Janz

Peter Januschke, Business Programming Languages Group, SAP AG

Holger Janz, Business Programming Languages Group, SAP AG

(complete bios appear on page 30)

If any of your SAP applications are web-enabled today (or soon will be!), Release 6.10 of the SAP Web Application Server (Web AS) offers some welcome enhancements that will make your life as an application developer much easier. The SAP Web AS supports direct Internet communication, and its new Internet Communication Manager (ICMAN) enables your SAP server to function as a web server on either your local network or the Internet. To take advantage of this new technology, you also need tools for developing web applications that provide direct and easy access to all SAP components. The solution is a combination of Business Server Pages (BSPs) and JavaScript. In this article, we explore how to use JavaScript to web-enable your SAP applications.

Why JavaScript?

If you're like most SAP customers, you're probably migrating at least some of your applications to the web. Online shops and online booking systems are common examples. As a result, you need to design and develop user interfaces in order to web-enable your existing programs. This migration is typically a two-step process:

- **Phase 1:** Make the existing code accessible by a web application, which requires new interfaces.
- **Phase 2:** Design the user interface.

Typically, different people are responsible for these phases. Developers who are familiar with the application perform Phase 1, while Phase 2 frequently requires other specialists (such as web page

designers). These designers often come from external companies or partners, and usually don't know anything about the ABAP language. JavaScript, which is commonly understood by web page designers, is a better option for making the application logic that is visible to the user comprehensible for user interface designers.

In addition, ABAP programmers sometimes want to run small scripts that are written in other languages.[1] Most often, you would choose this approach in order to exploit specific features that are either not available or difficult to mimic in ABAP. With the addition of JavaScript support, you can use the features of another language without having to use external tools.

How SAP Provides Support for JavaScript

A Business Server Page (BSP) is a web page in the SAP system. It can contain static components (HTML) and scripts, which are program fragments that retrieve data dynamically and format the page that is sent to the client when the web page is visited. You can, of course, use ABAP as a script language, which provides easy and direct access to SAP components. However, other developers, and in particular web user interface designers, are often not familiar with SAP's own programming language. In order to provide a modern and web-oriented alternative to ABAP, BSPs also support JavaScript as a script language.

Supporting JavaScript in BSPs, however, means more than simply ensuring that the development tools accept JavaScript as the script language. Runtime support is also necessary. For this reason, the SAP kernel now includes a complete, integrated JavaScript interpreter (Mozilla, JavaScript 1.5), which enables it to execute scripts written in JavaScript. In addition, you can also access ABAP objects and modules in JavaScript. In other words, you can link JavaScript variables to ABAP objects, access them, and (if the linked object is an instance of an ABAP class) invoke ABAP methods directly from JavaScript.

[1] Perl, for example.

This functionality is provided in an ABAP class named `CL_JAVA_SCRIPT`, enabling both web application developers and ABAP developers to use JavaScript programs in their applications. An interface is also provided for debugging JavaScript programs and for optimum reusability of compiled scripts.

Now let's examine the techniques for using JavaScript in SAP in detail. In the first part of this article, we look at how to create a BSP using JavaScript. In the second part of this article, we look at the challenges from a different perspective and discuss how to call JavaScript from your ABAP programs.

Part 1: Creating BSPs with JavaScript

BSPs are a means of providing web applications on an SAP application server.[2] The content, or layout, of a BSP consists of the following elements:

- Static text (HTML) that is transferred to the page and sent to the browser exactly as written
- Dynamic components that run when the page is visited and can dynamically format the page to be sent to the browser

Listing 1 shows the layout source code of a basic Business Server Page. As the article progresses, we'll show you how to construct simple pages like this one and add the elements that characterize more complex pages.

[2] For more information and tutorials on Business Server Pages, go to **http://help.sap.com**. Also refer to Karl Kessler's article "A Developer's Guide to Creating Powerful and Flexible Web Applications with the New Web Application Builder" in the January/February 2002 issue of this publication.

 ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Listing 1: Layout Source Code for a Simple BSP

```
<%@page language="javascript"%>

<html>
  <head>
    <title> First Demo </title>
  </head>
  <body>
    <% for ( i = 0; i < 10; i++ )
       {
    %>
         Hello World ! <br>
    <% } %>
  </body>
</html>
```

Listing 2: A Simple BSP with Static Text Only

```
<%@page language="javascript"%>

<html>
  <head>
    <title> Greeting </title>
  </head>
  <body>
    Hello World !
  </body>
</html>
```

When a BSP is activated, an ABAP method is created from the layout. This ABAP method is invoked whenever a URL request for the page is processed and ensures that the page to be returned to the browser is created from the static and dynamic components.[3]

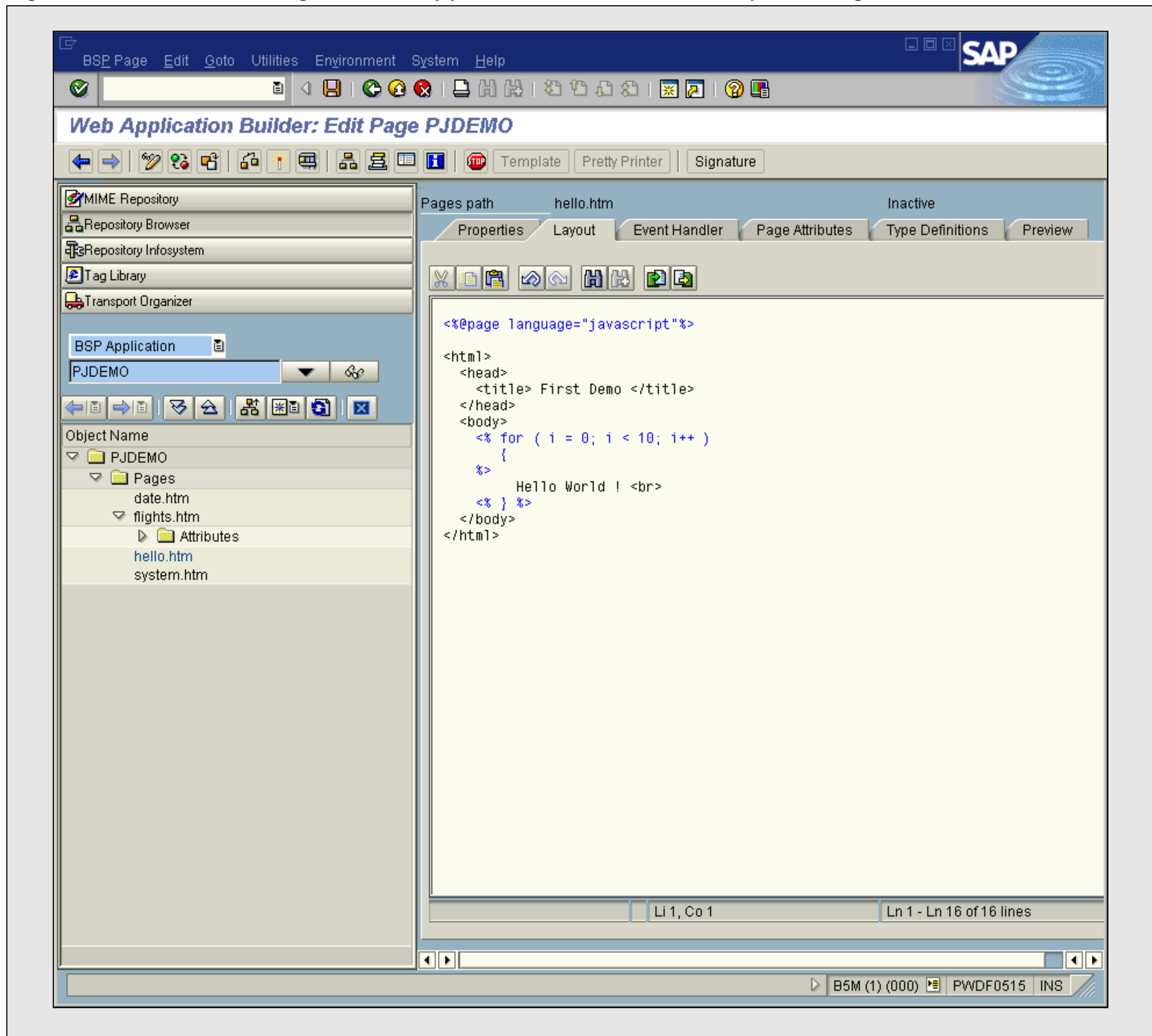
A BSP contains other components in addition to the layout. For example, you can define event handler methods that are executed if certain events occur when the BSP is processed. You can also define page attributes that can either be passed to the page as parameters when the page is navigated to, or used as global variables inside the BSP. In addition, instances of runtime objects provide special functionality in the layout and event handlers, such as for invoking methods from separate application logic. We only touch on these subjects here where it is necessary for understanding the examples. For more detailed information, refer to the BSP tutorials at **http://help.sap.com**.

[3] Note that this means that you shouldn't use client-side JavaScript within the scripts of a BSP (see also the sidebar on page 8).

Creating a Simple Business Server Page

First, let's start by creating a simple BSP (see **Listing 2**) that contains static text only — no scripts,

Figure 1 Using the Web Application Builder in the Object Navigator



page attributes, or event handlers. If you visit this page with a browser, you would see the message `Hello World !`. Our example begins with a page directive to inform the development environment that JavaScript is the script language for this page. Since this page does not contain a script, you could also specify ABAP by replacing `javascript` with `abap`.

You create and edit BSPs with the Web Application Builder (transaction SE80). **Figure 1** shows the Web Application Builder, which is a component of the Object Navigator.

Remember that a BSP must be part of a web application.[4] Therefore, before constructing a BSP, you must either create a new web application or know which existing one you want to use. Then go

[4] BSP-based web applications always include a set of Business Server Pages.

 ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.

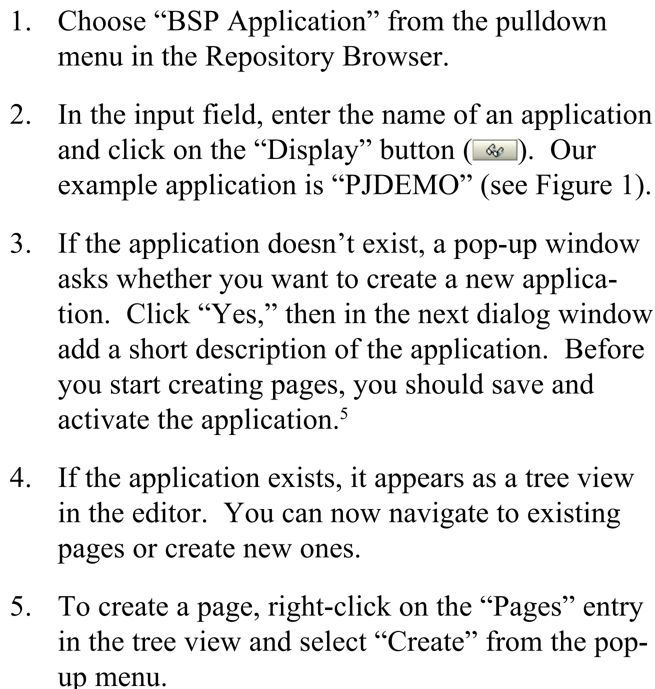
Listing 3: Controlling the Page Layout with a Script

```
<%@page language="javascript"%>

<html>
  <head>
    <title> Date </title>
  </head>
  <body>
    <% function number2Day( x )
        {
          switch ( x )
          {
            case 0 : return "Sunday";
            case 1 : return "Monday";
            case 2 : return "Tuesday";
            case 3 : return "Wednesday";
            case 4 : return "Thursday";
```

(continued on next page)

to the Object Navigator and follow these steps to create a BSP:

1. Choose "BSP Application" from the pulldown menu in the Repository Browser.
2. In the input field, enter the name of an application and click on the "Display" button (). Our example application is "PJDEMO" (see Figure 1).
3. If the application doesn't exist, a pop-up window asks whether you want to create a new application. Click "Yes," then in the next dialog window add a short description of the application. Before you start creating pages, you should save and activate the application.[5]
4. If the application exists, it appears as a tree view in the editor. You can now navigate to existing pages or create new ones.
5. To create a page, right-click on the "Pages" entry in the tree view and select "Create" from the pop-up menu.
6. The next dialog window shows the application name, and prompts you for the page name and a short description. A checkbox allows you to classify your page as a page fragment.[6]
7. You can now enter or modify the layout source code in the editor frame.[7]

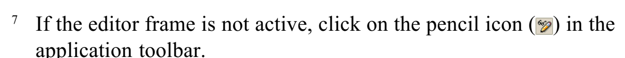
We assume that you already know how to use the page editor features to work with the layout source code. If you are just starting to use this tool, refer to the BSP tutorials and information sources mentioned earlier (see footnote 2).

Adding Scripts to a BSP

Now let's see how you could use JavaScript to control the page layout. For example, suppose you want to show today's day and date, and display a special message if it's the weekend. As shown in **Listing 3**,

[5] Of course, you could also do this after creating pages. When the application is activated, all of its pages are also activated. In an already-active application, you can activate each page individually.

[6] Page fragments are pieces of pages that are used by other BSPs.

[7] If the editor frame is not active, click on the pencil icon () in the application toolbar.

(continued from previous page)

```
            case 5 : return "Friday";
            case 6 : return "Saturday";
          }
        }

        var myDate = new Date( );
        var myDay  = myDate.getDay( );
     %>

     Today is <%= number2Day( myDay ) %>,
      <%= 1900 + myDate.getYear( ) %>
     -<%= 1 + myDate.getMonth( ) %>
     -<%= myDate.getDate( ) %>
     <br>

     <% if ( myDay == 0 || myDay == 6 )
        {
     %>
          Great !  It is the weekend !
     <% } %>
   </body>
 </html>
```

you would add a simple script written in JavaScript to handle this requirement.

As this example shows, you can use JavaScript in your BSP in two ways:

- Include entire program fragments, encapsulated by the character strings "`<%`" and "`%>`".
- Insert the results of individual expressions in static text. Although these expressions are framed by "`<%`" and "`%>`", they are also introduced by the equal sign ("="). You can see an example in Listing 3 above with the statement:

```
-<%= myDate.getDate( ) %>
```

You can also see in this example how to use scripts to control how the page looks. The sentence:

```
Great !  It is the weekend !
```

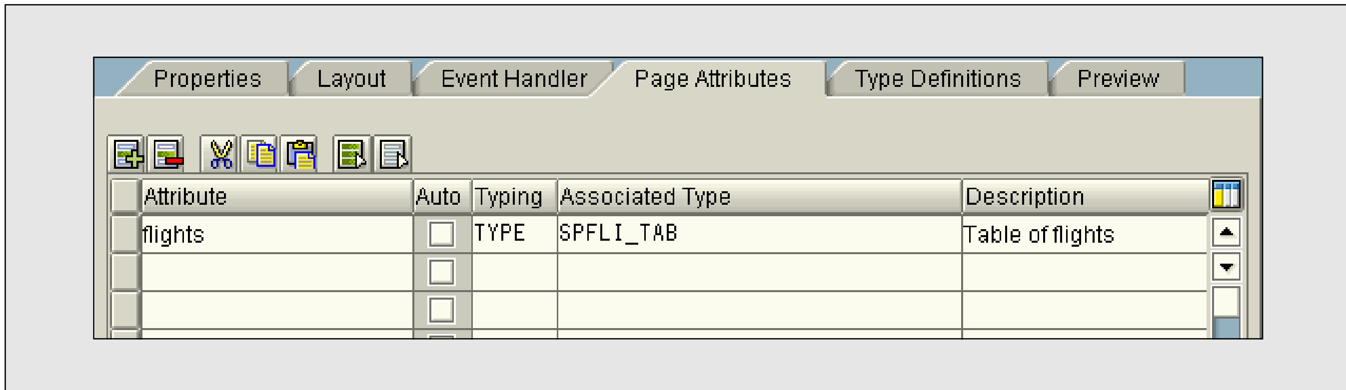
is only displayed in the browser if the condition `myDay == 0 || myDay == 6` is met. Of course, you can use any JavaScript language elements to

Server-Side vs. Client-Side JavaScript

Remember that you should only use server-side JavaScript in BSP scripts because the output page is always built on the server. For example, programming a dialog inside a BSP script is of no use. Clients only receive the complete HTML pages, which no longer contain the BSP scripts framed by "<%" and "%>". Obviously, you can add client-side JavaScript to the static components of a page, or dynamically create such scripts as a component of the output page.

 ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 2 *The "Page Attributes" Tab in the Object Navigator*



format the output. For example, you could use a loop to repeat the output part.

Displaying Data from a Table: Using Page Attributes

In addition to the page layout, a BSP also includes page attributes. Think of page attributes as variables that you can freely use in a BSP, especially in the page layout. This feature is particularly useful for passing parameters between web pages. Page attributes are also the means for sharing data between the layout and the event handlers.

Since an ABAP method is always generated from the page layout, page attributes are *always* ABAP fields. Nevertheless, you can address page attributes in JavaScript with the attribute name created in the Object Navigator. This option is possible because BSP page attributes are automatically linked to a JavaScript variable of the same name. The BSP runtime environment handles variable binding every time a BSP is visited. However, be aware that in JavaScript the name will be converted to lowercase. The attribute type (ABAP type) is mapped to the most suitable JavaScript object. For example, structures are linked as JavaScript objects with properties that have the same name as the ABAP structure components. In the same way, internal tables are mapped to array objects.

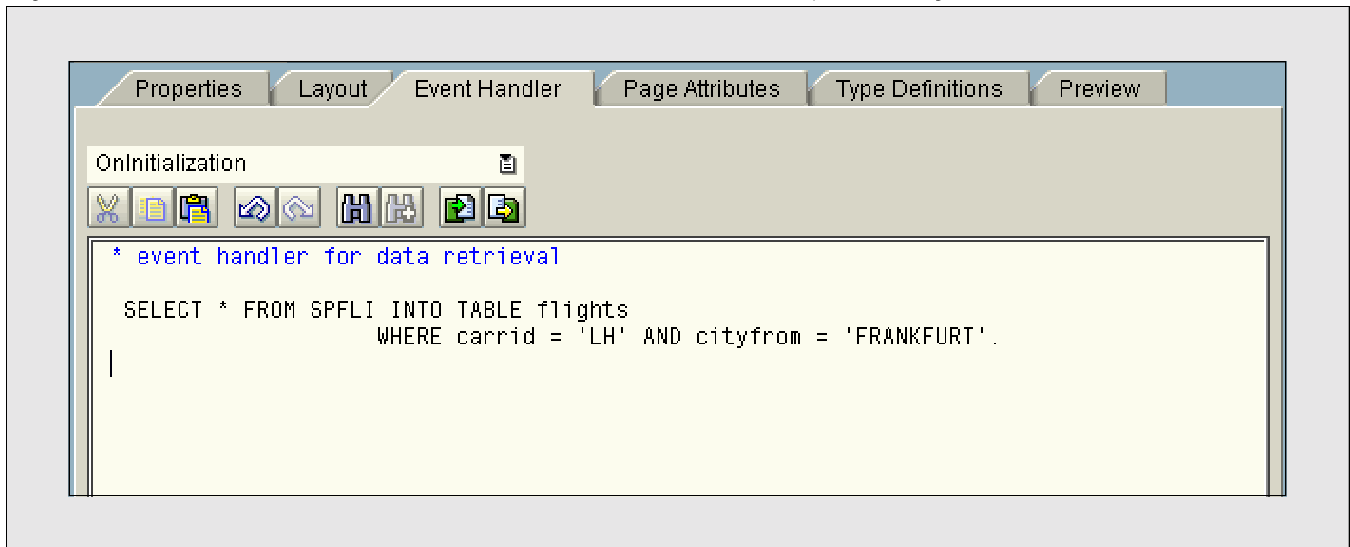
We distinguish between ordinary page attributes (which are used as global variables in BSPs) and automatic page attributes (which are externally passed to a page as parameters, such as from another BSP). The following example is confined to ordinary page attributes. We want to retrieve and use data within a single BSP, rather than pass data between several BSPs.

Now let's look at how you could use JavaScript with page attributes to retrieve and display more complex information on a page. In this example, we want to display flight data from the SPFLI table. Note that we have purposely kept this example simple. We want to show how you can use JavaScript in BSPs, rather than programming BSPs in general. Thus the BSP in our example has only three components: the page layout, a page attribute that stores the flight data, and an event handler for data retrieval.

Start by creating a page attribute on the relevant tab page in the Object Navigator, as shown in **Figure 2**. The data type "SPFLI_TAB" is a table type created in the Data Dictionary. Its row structure corresponds to the entries in the SPFLI table.

Next, to retrieve the data, go to the "Event Handler" tab page. Choose the "OnInitialization" event in the pulldown menu and enter the statements for retrieving the data. This event handler is invoked

Figure 3 *The "Event Handler" Tab in the Object Navigator*



once BSP processing begins, which is immediately after the server receives the request. As shown in **Figure 3**, the `SELECT` statement fills the page attribute `flights` with data from the `SPFLI` table.[8]

Now all you have to do is add the script shown in **Listing 4** to display the data as a table.

As you can see, you can use normal JavaScript notation to access the ABAP flights table. In this example, we access each individual table row via an index and retrieve components of the table row by specifying the component name. The binding mechanism of the extended JavaScript interpreter ensures that the ABAP table is correctly accessed. In the same way, the `length` property is added to the JavaScript array when it is linked. This property just gives the current number of rows in the table. It's also a read-only property; you can't assign a value to it. Later we'll take a detailed look at what operations are possible on ABAP data bound in JavaScript.

Figure 4 shows what you would see in your browser when visiting the page we just created.

[8] Event handler code is always written in ABAP.

Displaying Data from a Table: Calling an Application Class Method

All JavaScript-based BSPs that are called up from the server are automatically linked to several objects that you don't need to explicitly declare, such as the standard web request and response objects. We'll show you how to use these objects shortly, but first let's look at the application object.

You can bind an application class to each BSP application. It is simply a global class that provides the application logic required in the BSP application. When an application page is visited, an instance of the application class is created before the associated layout method is invoked, and a reference to that instance is passed to the layout method as a parameter. For JavaScript-based pages, this parameter is automatically bound to a variable called `application`. You can use this variable in JavaScript scripts to access all public attributes of the application class and invoke their public methods.

In the previous example, we retrieved the data in the "OnInitialization" event. Now let's change that example to retrieve the data by calling

 ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Listing 4: Accessing Page Attributes from Within JavaScript

```
<%@page language="javascript" %>
<html>
  <body>
    <table border=1>
      <tr><th>Carrier</th><th>Connection Code</th>
          <th>From</th>
          <th>City</th>
          <th>Departure Time</th><th>Arrival Time</th><th>Flight Time</th>
      <% for ( i = 0; i < flights.length; i++ )
         {
      %>
        <tr><td><%=flights[ i ].carrid %></td>
            <td><%=flights[ i ].connid %></td>
            <td><%=flights[ i ].cityfrom %></td>
            <td><%=flights[ i ].cityto %></td>
            <td><%=flights[ i ].deptime %></td>
            <td><%=flights[ i ].arrtime %></td>
            <td><%=flights[ i ].fltime %></td>
      <% } %>
    </table>
  </body>
</html>
```

Figure 4 ***The Resulting Page Viewed in a Browser***

Carrier	Connection Code	From	City	Departure Time	Arrival Time	Flight Time
LH	0400	FRANKFURT	NEW YORK	101000	113400	504
LH	0402	FRANKFURT	NEW YORK	133000	150500	504
LH	2402	FRANKFURT	BERLIN	103000	113500	65

a method of an application class directly in the BSP layout. Application classes are useful in two situations:

- Situations in which you want to separate the application logic from the web application. In other words, you provide an interface for accessing the application logic, but hide its implementation.
- Situations in which you want to share application logic in different applications. These applications can be other web applications, as well as pure ABAP applications.

In both situations, using an application class enables you to modify the logic without modifying the applications that use it.

Listing 5: ABAP Method to Retrieve Data from the Table SPFLI

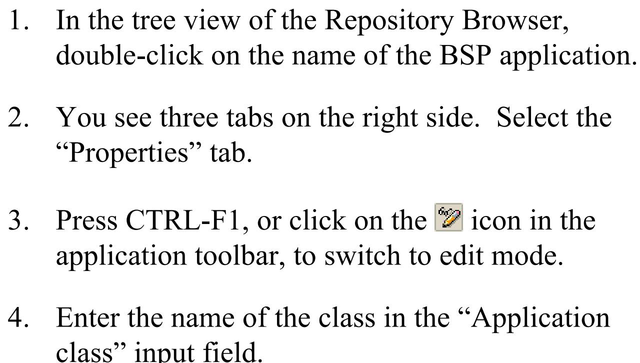
```
METHOD get_flights IMPORTING carrid      TYPE STRING
                             city        TYPE STRING
                   CHANGING  flightTable TYPE SPFLI_TAB.

  SELECT * FROM SPFLI INTO TABLE flightTable
                      WHERE carrid = carrid AND cityfrom = city.
ENDMETHOD.
```

We'll use the key values that were directly specified earlier for the `SELECT` statement as `IMPORTING` parameters for the method to be invoked. The requested table with flight data will be returned as a `CHANGING` parameter. Note that you can't use a `RETURNING` parameter, because you can't use assignment to store complex ABAP fields such as tables in a JavaScript variable.

First you need to create a global class `CL_MY_BSP_APP_CLASS`[9] in the Class Builder (transaction SE24) and implement a method that corresponds to the ABAP program fragment shown in **Listing 5**. This method fills the output parameter `flightTable` with data from the `SPFLI` table.

Before you can use the application logic, you need to introduce the class to the BSP application. Follow these steps in the Object Navigator:

1. In the tree view of the Repository Browser, double-click on the name of the BSP application.
2. You see three tabs on the right side. Select the "Properties" tab.
3. Press CTRL-F1, or click on the  icon in the application toolbar, to switch to edit mode.
4. Enter the name of the class in the "Application class" input field.

[9] Of course, you can choose your own class name.

5. Save and activate the application.
6. Navigate back to the BSP.

We again use the `flights` page attribute to access the flight data, since it is automatically bound in the JavaScript context.

Finally, you have to insert a call to the `get_flights` method in the BSP before the HTML tag `<table border=1>`. For this purpose, you use the `invoke` instance method, which is automatically added to the application object. As shown in **Listing 6**, as its first parameter, this function expects the name of the ABAP method as a character string. Pairs of parameter names and values follow, with the parameter names specified first, and designated as character strings.

For ABAP developers, the JavaScript method invocation in Listing 6 corresponds to the ABAP statement shown in **Listing 7**.

> ✓ ***If You're Trying This Example…***
>
> *When you're following this example yourself, first add an arbitrary public attribute to the application class. Otherwise, the application object cannot be automatically bound to the JavaScript interpreter.*

 ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Listing 6: Invoking the Application in JavaScript

```
<%  application.invoke( "get_flights",
                        "carrid", "LH",
                        "city", "FRANKFURT",
                        "flighttable", flights );
%>
```

Listing 7: Invoking the Application in ABAP

```
CALL METHOD application->get_flights EXPORTING carrid      = 'LH'
                                               city        = 'FRANKFURT'
                                     CHANGING  flighttable = flights.
```

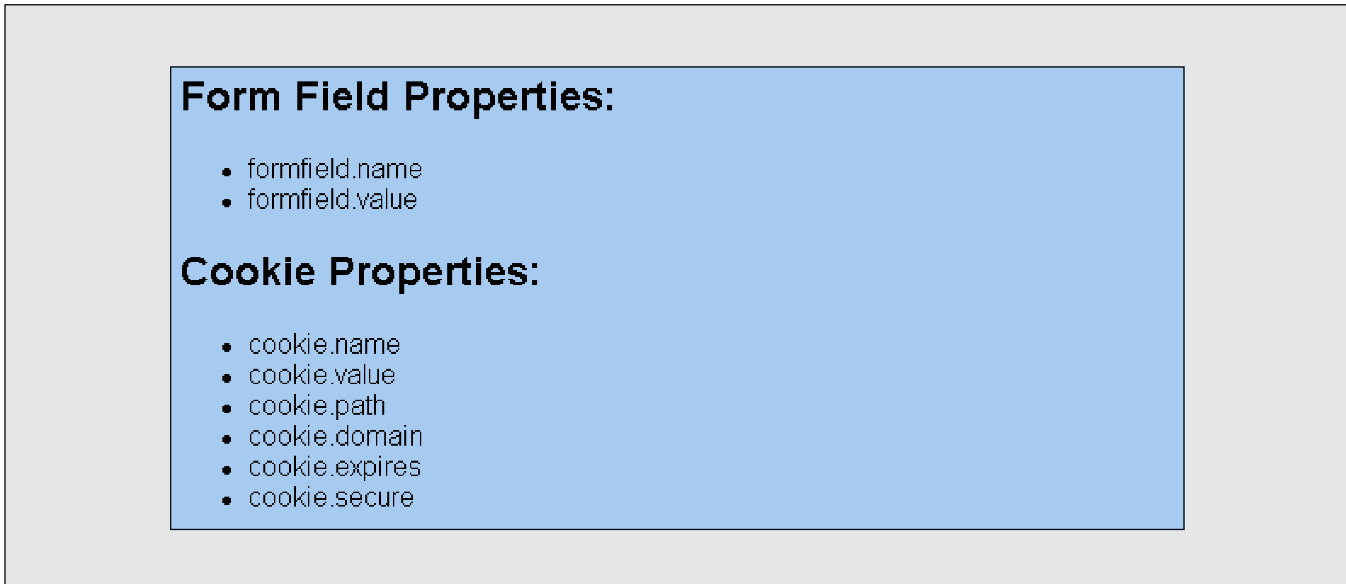
Listing 8: Accessing the Request Object

```
<%
   if ( request.formfields.length > 0 )
   {
     headingLine( 'Form Field Properties:' );
     listStart( );
     for ( prop in request.formfields[ 0 ] )
       listLine( 'formfield.', prop );
     listEnd( );
   }

   if ( request.cookies.length > 0 )
   {
    headingLine( 'Cookie Properties:' );
    listStart( );
    for ( prop in request.cookies[ 0 ] )
      listLine( 'cookie.', prop );
    listEnd( );
   }
%>
```

The example in **Listing 8** shows you a way to access information from the request object. Here, we use some functions to format the HTML output in the output page (`headingLine`, `listStart`, `listLine`, and `listEnd`). For the sake of simplicity, let's assume they have already been defined.

Figure 5 *The Resulting Page Viewed in a Browser*



Listing 9: Querying System Settings in a BSP

```
<%@page language="javascript"%>
<html>
  <body class="bspBody1">
     User : <%=sy.uname%> <br>
     System : <%=sy.sysid%> <br>
     Release: <%=sy.saprl%> <br>
     Database: <%=sy.dbsys%> <br>
     Server: <%=sy.host%> <br>
     Time: <%=sy.uzeit%> <br>
  </body>
</html>
```

In Listing 8, we also show some available properties for form fields and cookies. As you can see, you can easily use the ABAP object attributes in JavaScript. **Figure 5** shows the browser view.

Accessing System Fields

In addition to accessing the objects you have seen so far, you can also access the system fields from the ABAP runtime environment (structure `SY` of type `SYST`) in JavaScript. This structure is bound automatically under the JavaScript name `sy`. In this way, you can also query system settings in a BSP. For example, this technique enables you to add system information to the output page.[10] **Listing 9** shows how easy this is to accomplish.

[10] For example: "This page was created for user JANUSCHKE on system B5M."

 ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.

When viewed in a browser, the page is displayed as follows:

```
User : JANUSCHKE
System : B5M
Release: 610
Database: MSSQL
Server: PWDF0515
Time: 175420
```

ABAP/JavaScript Differences and Limitations

As you have seen, you can easily access many things in the SAP world by binding ABAP objects to the JavaScript interpreter. Nevertheless, remember that there are many differences between ABAP objects and JavaScript objects. Consequently, you also need to be aware of limitations to accessing ABAP data objects from JavaScript.

ABAP is a typed language that supports many different data types. You can declare data types in ABAP programs, as well as parameterize them according to the needs of your application. For example, internal tables are a powerful ABAP data type that allow you to store and manage many objects of the same type. ABAP supports three basic forms of internal tables — standard tables (as shown in our examples), sorted tables, and hash tables where the data is accessed via a hash code rather than an index. However, JavaScript only supports one type of array. Therefore, it shouldn't surprise you that sorted tables and hash tables cannot be bound to JavaScript.

You can also encounter problems with more basic data types. Apart from objects, JavaScript only distinguishes character strings and numerical values. In some cases, however, no distinction is made between character strings and numbers, or a numerical value is implicitly converted to a character string. For example, let's examine the following ABAP data declaration:

```
DATA number TYPE I VALUE 10.
```

Assume that this object is bound to JavaScript under the same name. In this case, the following script:

```
<%=number+"50"%>
```

would not yield the expected value of 60, but rather a value of 1050. JavaScript interprets `"50"` as a character string and regards the entire expression as a concatenation of character strings.

ABAP also supports a data type for decimal arithmetic that does not have an equivalent in JavaScript, and is therefore bound as a string object. You could encounter the same problem when adding values. You can, however, avoid this situation by explicitly specifying the value of the bound variable as a number object:

```
DATA number TYPE P DECIMALS 2
  VALUE '10.55'.

<%=Number(number)+50%>
```

Yet another problem occurs with ABAP objects whose identifier matches a reserved JavaScript keyword such as `byte`, `class`, or `null`. Although these objects can be bound to JavaScript, using them would cause the JavaScript interpreter to report a syntax error.

When dealing with complex data objects such as structures, tables, and object references, you may commonly truncate long access paths by assigning sub-objects to JavaScript objects, especially to shorten runtime. However, be careful when ABAP objects are involved. Access to sub-objects is only ensured if all ABAP objects involved remain unchanged when accessed — that is, if the complete access path to the sub-object still exists. Consider, for example, a method call. If objects belonging to the access path are released, any subsequent access via JavaScript objects may have unexpected effects that could lead to program termination in the ABAP

Listing 10: Unsafe Access Paths

```
TYPES:  BEGIN OF typ_struct,
          integer      TYPE I,
          character(1) TYPE C,
        END OF typ_struct.

CLASS cl_myClass DEFINITION.
  PUBLIC SECTION.
    DATA myTab TYPE table OF typ_struct.
    METHODS myMethod.
ENDCLASS.

CLASS cl_myClass IMPLEMENTATION.
  METHOD myMethod.
    CLEAR myTab INDEX 3.
  ENDMETHOD.
ENDCLASS.

DATA sample TYPE REF TO cl_myClass.
CREATE OBJECT sample.
DO 3 TIMES.
  APPEND INITIAL LINE TO sample->myTab.
ENDDO.

<%
   // step 1a: slow access
   sample.myTab[ 2 ].integer = 12;

   // step 1b: faster access, but possibly unsafe
   var ref_struc = sample.myTab[ 2 ];
   ref_struc.integer = 15;

   // step 2: call ABAP method
   myRef.invoke( "myMethod" );

   // step 3: causes a runtime error
   ref_struct.integer = 1;
%>
```

runtime environment. Let's look at an example of this behavior in **Listing 10**.

In the coding below the implementation of `cl_myClass`, we first create an instance of this class. Afterward, we append three initial lines to the attribute `myTab`, which is an internal table. Now we access this table from within JavaScript.

The assignment in "step 1a" requires two accesses to ABAP objects — `sample.myTab[ 2 ]` and `myTab[ 2 ].integer`. Alternatively, the assignment in "step 1b" enables faster access, especially if you need to repeatedly access the components of `myTab[ 2 ]`. First you store a reference to `sample.myTab[ 2 ]` in the JavaScript variable `ref_struc`. Once you've done this, you don't need

 ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 6 *Using JavaScript in BSPs*

Technique	Purpose
JavaScript control structures (conditional statements, loops, etc.)	Construct the page layout dynamically, or define and use small utility functions.
JavaScript expressions <%=expression%>	Dynamically insert values into the static elements of a Business Server Page.
Page attributes	Retrieve and supply data.
Event handlers	Perform developer-defined actions at specific points during page processing, such as data retrieval.
Application classes	Provide sophisticated application logic, such as complex data retrieval and preparation, or calculations.
Application objects	Access the application logic from within JavaScript.
Request and response objects	Access typical web application elements from within JavaScript, such as cookies and form fields.
ABAP system fields	Access system information from within JavaScript, such as user ID or server name.
Data binding limitations	Understand and avoid things that either don't work as expected, or don't work at all.

to access the ABAP object in order to access structure components. Instead, you can directly access the substructure using the reference in `ref_struc`.

After choosing slow (step 1a) or faster (step 1b) access, the ABAP method `myMethod` is invoked and deletes the line of `myTab` that corresponds to `sample.myTab[ 2 ]`.[11] The access path to `sample.myTab[ 2 ]` then disappears from the JavaScript interpreter, and the subsequent assignment to `ref_struc.integer` will lead to program termination.

A Checklist of Techniques Learned

As a review of what you have learned in this section, **Figure 6** summarizes the key techniques for using JavaScript in BSPs.

[11] Numbering of table lines differs by 1 between ABAP and JavaScript.

> ✓ ***Other Tools for Developing BSPs***
>
> *You can use the Object Navigator for developing BSPs with other tools besides the editor. For example, it also provides tools for maintaining MIME objects and managing text to be stored in the Online Text Repository. Obviously, you can also debug BSPs — even the best developers sometimes make mistakes that would be difficult to detect without utilities. For more detail, see the online BSP tutorials and Karl Kessler's article in the January/February 2002 issue.*

Part 2: Calling JavaScript from Your ABAP Programs

You now understand how to access ABAP objects from JavaScript. It's time to put on an ABAP

developer's hat and explore how you can access JavaScript directly from your ABAP programs. As introduced earlier, an interface between ABAP and the JavaScript interpreter enables you to create and execute JavaScript programs from any ABAP programs. You can share data between ABAP and JavaScript, as well as debug JavaScript programs. Plus, compiled JavaScript scripts can be stored in internal format,[12] which reduces runtime when scripts are executed again since they don't need recompiling.

The secret behind the JavaScript interpreter is the ABAP class CL_JAVA_SCRIPT. It's particularly useful for generating layout methods for JavaScript BSPs. JavaScript BSP developers don't really need to know about the CL_JAVA_SCRIPT class, since it's used behind the scenes. Thus the techniques we explore in the second half of this article are mainly of interest to ABAP developers who want to use and manage their own JavaScript programs. We also share some insight into binding ABAP data objects.

Creating a JavaScript Runtime Context

If you want to use the JavaScript interpreter in your ABAP program, the first thing you need is an instance of the class CL_JAVA_SCRIPT. You can then handle all further activities in your program by calling the instance methods of this object. Therefore, an instance of CL_JAVA_SCRIPT is also referred to as a JavaScript context. Your program *must* include the following program fragment to create this context:

```
DATA js_processor TYPE REF TO
  CL_JAVA_SCRIPT.
js_processor =
  CL_JAVA_SCRIPT=>CREATE( ).
```

Invoking the CREATE class method is the only way of generating instances of CL_JAVA_SCRIPT. Since the class was declared with the addition CREATE PRIVATE, instances cannot be created with the CREATE OBJECT statement. From now on, our examples assume that a JavaScript context has already been created with the name js_processor, rather than repeating this code each time. Of course, you can also create several contexts within the same program.

[12] A JavaScript interpreter doesn't interpret the text of a JavaScript program. Instead, it first translates it into byte code.

In a JavaScript context, two attributes are set for almost all of its instance method calls. After invoking the method, you can check whether the operation was successful. The CREATE method is an exception, since it returns an initial reference in the event of an error instead of setting these attributes. For all other methods, you can use these attributes to check error conditions:

- The attribute LAST_CONDITION_CODE of type I (integer) contains a value that indicates whether an error has occurred. CL_JAVA_SCRIPT provides a set of predefined constants for use as comparison values. Use the Class Builder (transaction SE24) to look up these values.
- If an error occurs, the attribute LAST_ERROR_MESSAGE contains an error message describing the error in more detail.

The following sections contain examples that demonstrate how to evaluate these attributes.

Methods for Working with JavaScript Programs

JavaScript programs and their results are represented as character strings (ABAP type STRING). **Listing 11** shows the methods you can use in your ABAP programs for working with JavaScript.

The COMPILE method translates a JavaScript program into byte code and stores it in the JavaScript context under a specified name. You can execute a compiled program and obtain its result using the EXECUTE method. The DESTROY method deletes a program from the context. Finally, the EVALUATE method combines these three steps in a subroutine to form a single step. Let's examine the multi-step method first by looking at how to sequentially

 ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Listing 11: ABAP Methods for Working with JavaScript

```
METHODS:
  COMPILE  IMPORTING script_name TYPE STRING
                     script      TYPE STRING,

  EXECUTE  IMPORTING script_name TYPE STRING
           RETURNING result      TYPE STRING,

  DESTROY  IMPORTING script_name TYPE STRING.

  EVALUATE IMPORTING java_script TYPE STRING
           RETURNING result      TYPE STRING,
```

Listing 12: Compile, Execute, and Delete a JavaScript Program

```
DATA: myProgram  TYPE STRING,
      myProgName TYPE STRING,
      myResult   TYPE STRING.

myProgram = 'myDate = new Date( ).getYear( );';
myProgName = 'year.js'.

* step 1: compilation
CALL METHOD js_processor->COMPILE( script_name = myProgName
                                   script      = myProgram  ).

IF js_processor->LAST_CONDITION_CODE <> CL_JAVA_SCRIPT=>CC_OK.
  WRITE js_processor->LAST_ERROR_MESSAGE.
  EXIT.
ENDIF.

* step 2: execution
myResult = js_processor->EXECUTE( myProgName ).

IF js_processor->LAST_CONDITION_CODE <> CL_JAVA_SCRIPT=>CC_OK.
  WRITE js_processor->LAST_ERROR_MESSAGE.
ELSE.
  WRITE myResult.
ENDIF.

* step 3: destruction
js_processor->DESTROY( myProgName ).
```

compile, execute, and delete a JavaScript program within an ABAP program (see **Listing 12**).

This example starts by saving a small JavaScript program in the `myProgram` variable as `year.js`.

Listing 13: Single-Step JavaScript Program Execution

```
DATA: myProgram TYPE STRING,
      myResult  TYPE STRING.

myProgram = 'myDate = new Date( ).getYear( );';
myResult = js_processor->EVALUATE( myProgram ).

IF js_processor->LAST_CONDITION_CODE <> CL_JAVA_SCRIPT=>CC_OK.
  WRITE js_processor->LAST_ERROR_MESSAGE.
ELSE.
  WRITE myResult.
ENDIF.
```

The ABAP program then compiles the JavaScript program, executes it, and finally deletes it.

In "step 1," the ABAP program compiles the JavaScript program by calling the `COMPILE` method. The JavaScript name to use for storing the script is specified, along with the script itself. The `IF` statement then checks the `LAST_CONDITION_CODE` attribute to see if an error has occurred, displays the error message from the `LAST_ERROR_MESSAGE` attribute if necessary, and ends the ABAP program execution.

If the JavaScript program is successfully compiled, "step 2" executes this program by calling the `EXECUTE` method. You pass only the name of the script (specified in step 1) to this method. The example then checks for errors and, if any exist, outputs an error message. This time though, we don't end the ABAP program execution, since we stored the script in the JavaScript context and want to delete it first. If no errors occurred, we output the result of the JavaScript program. The return value of the `EXECUTE` method is always the result of the final statement executed in the JavaScript program. In this example, the program outputs the current year.

In "step 3," the compiled JavaScript program is deleted from the JavaScript context by invoking the `DESTROY` method. It expects only the program name as a parameter.

The only reason to carry out these three steps separately is if you want to execute the compiled JavaScript program several times, either at several points in the ABAP program or within a loop. If you only want to execute the JavaScript program once, use the `EVALUATE` method instead. It executes the three steps above in sequence, which would significantly simplify and shorten our example. As you can see in **Listing 13**, using `EVALUATE` reduces the number of method calls. Plus, you don't need to specify a name for the JavaScript program.

Binding ABAP Objects for JavaScript Access

When you create JavaScript-based BSPs, ABAP data objects can be bound to the JavaScript interpreter automatically. However, in ABAP programs, you must bind the objects yourself. Two methods — `BIND` and `BIND_INSTANCE` — are provided for this purpose (see **Listing 14**).

In general, you use the `BIND` method to include ABAP objects in a JavaScript context. The second method, `BIND_INSTANCE`, is a variant of `BIND` that is specifically provided for class instances. We'll look at the difference later, after examining the parameters.

You can include objects in the JavaScript context in two ways, either as a global variable or as a

    ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Listing 14: Methods for Binding ABAP Objects to JavaScript

```
METHODS:
 BIND          IMPORTING name_obj   TYPE STRING
                         name_prop  TYPE STRING
               CHANGING  data       TYPE ANY,

 BIND_INSTANCE IMPORTING name_obj   TYPE STRING
                         name_prop  TYPE STRING
               CHANGING  object_ref TYPE REF TO OBJECT.
```

Listing 15: Comparison of Binding Methods

```
DATA:  int    TYPE I VALUE 5,
       float  TYPE F VALUE '4.5',
       result TYPE STRING.

CALL METHOD js_processor->BIND EXPORTING name_obj  = ''
                                         name_prop = 'number'
                               CHANGING  data      = int.

CALL METHOD js_processor->BIND EXPORTING name_obj  = 'abap'
                                         name_prop = 'number'
                               CHANGING  data      = float.

result = js_processor->EVALUATE( 'number + abap.number;' );
```

Binding to a global variable

Binding to a root object

sub-object of a root object. The following parameters are significant:

- The `name_obj` parameter controls how the object is bound. If, when invoked, an empty character string is passed as a parameter, the ABAP object is bound to a global variable. Otherwise, it is bound as a sub-object to the root object `name_obj`.

- The `name_prop` parameter determines the name under which you access the ABAP object in JavaScript. You must specify a valid JavaScript identifier. Note that uppercase/lowercase notation is important.

- The combination of `name_obj` and `name_prop` determines the access path in JavaScript.

- The `data` or `object_ref` parameter contains the object to be bound. You may be wondering why this is a `CHANGING` parameter and not an `IMPORTING` parameter. The reason is that the `BIND` method transfers the properties of the passed data object to the JavaScript object. An `IMPORTING` parameter is, however, write-protected, so you couldn't assign any values to the bound object.

The example in **Listing 15** demonstrates the

Listing 16: Binding System Fields for JavaScript Access

```
CALL METHOD js_processor->BIND EXPORTING name_obj  = ''
                                         name_prop = 'sy'
                               CHANGING  data      = sy.
```

difference between binding to a global variable and binding to a root object.

When you invoke `BIND` the first time, the ABAP field `int` is bound to the global JavaScript variable `number`. When you invoke it the second time, the `float` field is bound to the root object `abap` as the property `number`. When the JavaScript program is executed, it calculates the total from `int` and `float` and returns the result `9.5`.

It may seem tempting to always include ABAP objects as global variables in a JavaScript context, especially since you can then create as many global variables as you want. Nevertheless, use this type of binding sparingly. It restricts the namespace for pure JavaScript variables, which increases your risk of naming conflicts.

The advantage of using root objects is that by choosing the right names, you can easily make JavaScript programs more comprehensible. The name `abap.number` indicates that a bound ABAP object is used, as opposed to just `number`.

Remember that a newly created JavaScript context does not contain any automatically bound objects, such as `sy`. These objects, which are available in BSPs, are bound at runtime by the layout method generated in the BSPs. If you want to access `sy` fields in JavaScript,[13] you must handle the binding yourself, as shown in **Listing 16**.

[13] Outside of Business Server Pages.

Accessing ABAP Structures, Substructures, and Internal Tables

You now understand how binding works when you want to access JavaScript from your ABAP programs. But you still need to understand how ABAP data types are mapped to JavaScript objects when the ABAP objects are bound.

The explanation is easiest for the basic data types. Types `F` (float) and `I` (integer) are represented in JavaScript as numbers. The other numerical data types `N` (number) and `P` (packed) are converted to character strings. Otherwise, you could encounter problems such as loss of accuracy. (This procedure has the disadvantage already described of leading to unexpected results when you use the "+" operator.) The basic data types `C` (character), `D` (date), `T` (time), `X` (heXadecimal), `STRING`, and `XSTRING` are also mapped to character strings.

However, an ABAP structure is bound as a JavaScript object that contains all ABAP structure components as properties, with lowercase component names. Substructures are stored as separate objects. You can only make assignments to a bound ABAP structure in JavaScript by assigning each basic property individually. You can't assign an entire JavaScript object to an ABAP structure in a single step. Similarly, you can't add any new properties to a bound ABAP structure in JavaScript. You have already seen examples of using ABAP structures in the BSP section. Here, we present another example (see **Listing 17**) of ABAP structure assignments in

 www.SAPpro.com ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Listing 17: Assignments to ABAP Structures in the JavaScript Context

```
DATA: BEGIN OF address,
        first_name  TYPE STRING,
        family_name TYPE STRING,
        BEGIN OF city,
          zipcode(5) TYPE N,
          name       TYPE STRING,
        END OF city,
      END OF address,
      source TYPE STRING.

CALL METHOD js_processor->BIND EXPORTING name_obj  = 'abap'
                                         name_prop = 'addr'
                               CHANGING  data      = address.

CONCATENATE 'abap.addr.first_name   = "Han";        '
            'abap.addr.family_name  = "Solo";       '
            'abap.addr.city.zipcode = 64283;        '
            'abap.addr.city.name    = "Horse Town"; '
       INTO source SEPARATED BY CL_ABAP_CHAR_UTILITIES=>CR_LF.

js_processor->EVALUATE( source );

WRITE: / address-first_name,   address-family_name,
       / address-city-zipcode, address-city-name.

js_processor->EVALUATE( 'abap.addr = "Hello!";' );

WRITE js_processor->LAST_ERROR_MESSAGE.
```

First JavaScript program

Second JavaScript program

the JavaScript context. It shows how to change components of an ABAP structure in JavaScript, and also illustrates a key concept: assignment of a value to an entire structure produces an error.

In this example, the first JavaScript program fills the `address` structure components with values. It then outputs these values in the ABAP program, after calling the `EVALUATE` method. However, executing the second JavaScript program leads to an error because we are attempting to assign a value to an ABAP structure.

Internal standard tables are available in JavaScript in the form of array objects. Each array object has a `length` property that specifies the current number of rows in the table, and an `append` method without parameters, with which you can add initial rows to a table. In contrast to ABAP tables, array element numbering begins at 0 instead of 1. As for ABAP structures, you can't extend bound internal tables to include other properties, nor can you assign a value to a table as a whole. You have already seen examples of using internal tables in JavaScript in the BSP section, so it's not necessary to provide

any additional examples here. However, remember that hashed and sorted tables cannot be bound to JavaScript.

Calling ABAP Methods

ABAP object references are bound to the corresponding JavaScript objects so you can access public attributes and invoke public instance methods. Starting with Release 6.20 of the SAP Web AS, you can also access static attributes and invoke static methods. As for structures and tables, you can't extend bound object references in JavaScript to include further properties. The static type[14] of the bound object reference determines which attributes and methods you can access. Due to polymorphism, the bound object reference type can change dynamically in assignments in ABAP. However, this narrowing or widening of referenced objects in ABAP has no effect on JavaScript access. You can't access attributes that don't exist in the referenced object.

As mentioned earlier, you can call ABAP methods with the JavaScript method `invoke`. This method is added to each bound object. However, remember that you can only invoke the public methods of the bound instance. In addition, calling a method has the following constraints:

- The method's formal parameters, to which an object of basic type (including `SAPAbapMethParam`[15]) must be passed, and must be typed in full.
- The method can have a maximum of 32 formal parameters.
- The method can't contain any dialogs. It must be possible to execute the method in the background.
- You can't catch ABAP exceptions based on an exception class in JavaScript. You can, however, handle them in a `TRY` block in ABAP.

[14] This is the type that was specified in the declaration of the ABAP object.

[15] We'll discuss this special class object in more detail in a moment.

When calling methods, you always specify actual parameters as pairs of parameter names (these *must* be lowercase) and values (as shown in previous examples). The parameter sequence is therefore irrelevant. You don't have to specify a `RETURNING` parameter with a basic type (numbers or character strings) as a parameter for the `invoke` call. Instead, you can assign it to a JavaScript variable as the result of `invoke`. However, you must specify a `RETURNING` parameter with a complex data type (structure, table, or reference) as a parameter when `invoke` is called. Regardless of type, parameters of a basic type are always passed by value. If you want to pass a value such as this by reference, you must use an object of type `SAPAbapMethParam` that has exactly one property of type `String`. Furthermore, a type check is carried out. The types of actual and formal parameters must be compatible, otherwise the JavaScript interpreter terminates program execution with an error.

You have already seen some examples of how to access attributes and invoke methods in the BSP section. In contrast, **Listing 18** shows you how to pass parameters by reference.

In this example, the first program passes a simple variable without properties to the `set_value` function. Because this variable is always passed by value, the assignment within `set_value` does not change `js_object`. Therefore, the `WRITE` statement outputs the initial value `Init` of `js_object`. However, the second program passes an object to `set_value`. Because this object is passed by reference, the original value `Init` of the single `js_object` property is replaced by a random number, which is output in the final `WRITE` statement.

This behavior applies to all JavaScript objects, although only when calling JavaScript functions. You could use any JavaScript object, as shown in this example, with the same effect. However, you can't pass pure JavaScript objects to ABAP methods.

 ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Listing 18: Passing Parameters by Reference

```
DATA:  source TYPE STRING,
       result TYPE STRING.

CONCATENATE 'function set_value( object ) '
            '{  object = Math.random( ); }'
            '                             '
            'var js_object = "Init";      '
            '                             '
            'set_value( js_object );      '
            'js_object;                   '
       INTO source SEPARATED BY CL_ABAP_CHAR_UTILITIES=>CR_LF.

result = js_processor->EVALUATE( source ).

WRITE / result.

CONCATENATE 'function set_value( object )                  '
            '{  object.value = Math.random( ); }           '
            '                                              '
            'var js_object = new SAPAbapMethParam( "Init" );'
            '                                              '
            'set_value( js_object );                       '
            'js_object.value;                              '
       INTO source SEPARATED BY CL_ABAP_CHAR_UTILITIES=>CR_LF.

result = js_processor->EVALUATE( source ).

WRITE / result.
```

First program

Second program

In order to pass parameters of basic type by reference, you must always use the special `SAPAbapMethParam` class object to call ABAP methods. Otherwise, the `invoke` method can't recognize that the parameter must be passed by reference. We'll look at another example of passing references shortly, in connection with handling exceptions at method calls.

Remember that you can also use the `BIND_INSTANCE` method for binding ABAP object references to JavaScript. While the `BIND` method binds the object reference, the `BIND_INSTANCE` method binds the instance to which the reference refers. Since it creates a snapshot of the reference to the instance, the type[16] of the referenced object is decisive. In JavaScript, you can only access attributes and methods that are visible in the object when it is bound. Post-binding changes to the ABAP object reference have no effect on the JavaScript object. In most situations, you can handle bound instances in the same way as bound object references. However, you can't pass the bound instance as an actual parameter when calling an ABAP method via `invoke`, because the (no longer available) reference to the instance is required. As with BSPs, the

[16] This is the instance type when it is bound. Remember that due to polymorphism, the instance type can change dynamically during program execution.

Listing 19: Using the EXCEPTION Parameter to Handle ABAP Exceptions

```
CLASS myClass DEFINITION.
  PUBLIC SECTION.
    METHODS myMethod EXCEPTIONS myExc.
ENDCLASS.

CLASS myClass IMPLEMENTATION.
  METHOD myMethod.
    RAISE myExc.
  ENDMETHOD.
ENDCLASS.

DATA: myRef  TYPE REF TO myClass,
      source TYPE STRING,
      result TYPE STRING.

CALL METHOD js_processor->BIND EXPORTING name_obj  = 'abap'
                                         name_prop = 'ref'
                               CHANGING  data      = myRef.

CREATE OBJECT myRef.

CONCATENATE 'var exc = new SAPAbapMethParam( );               '
            'abap.ref.invoke( "myMethod", "EXCEPTION", exc );'
            'exc.value;                                       '
       INTO source SEPARATED BY CL_ABAP_CHAR_UTILITIES=>CR_LF.

result = js_processor->EVALUATE( source ).
WRITE / result.
```

application object `application` is bound as an instance. Therefore, the same restriction applies.

Exception Handling

No discussion of calling JavaScript from ABAP would be complete without at least briefly touching on exception handling at method calls. We have already said that you can't handle class-based exceptions in JavaScript. However, this restriction doesn't apply to classical ABAP exceptions that you declare with `EXCEPTIONS` when the method is defined. These types of exceptions provide the exception name as a character string, which can be used for case distinction during exception handling. The `invoke` method allows you to catch these exceptions. For this purpose, you specify an `EXCEPTION` parameter in the method call and pass to it an object of the predefined `SAPAbapMethParam` type. The example in **Listing 19** shows you how to handle exceptions in this way.

Here, the method `myMethod` assigns the value `MYEXC`[17] to the JavaScript variable `exc`, which is then returned and output as the result of the JavaScript program.

[17] ABAP uses these names in uppercase.

 ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Destroying a JavaScript Context

To conclude the topic of binding, note that you can't undo bindings for individual objects. The only way to release bound JavaScript variables is to explicitly delete the entire context, which you can do in two ways:

- Invoke the ABAP_DESTRUCTOR method:

  ```
  js_processor->ABAP_DESTRUCTOR( ).
  ```

  This call deletes the JavaScript context at the point at which it takes place. This means that the ABAP object is invalidated and the memory held by the JavaScript context is released. You should also explicitly release the instance that js_processor refers to with a CLEAR statement. The memory space occupied by the instance is then released during the next garbage collection run.

- Release the js_processor instance with a CLEAR statement. Both the JavaScript context and the CL_JAVA_SCRIPT object are deleted during garbage collection.

If you don't use either method, the context and the object are automatically released by the normal garbage collection process.

Finally, avoid binding ABAP objects in a JavaScript context that exists beyond the lifetime of the ABAP object. For example, it's useless to access a local field of a FORM subroutine in a JavaScript context that still exists after program execution returns from the subroutine, because the local field of the subroutine is no longer available. The best practice is to only bind globally accessible ABAP objects, or to use JavaScript contexts locally within subroutines.

Other Useful CL_JAVA_SCRIPT Features

✓ The CL_JAVA_SCRIPT class offers an extensive interface for debugging JavaScript programs. Since this interface is only interesting if you intend to write your own JavaScript debugger, we do not describe it in detail here. You can find more information in the ABAP online documentation (ABAPHELP).

✓ The methods FREEZE and THAW retrieve the byte code of a compiled JavaScript program from the JavaScript context, and reinsert stored byte code in a JavaScript context. They are useful for storing the internal format of JavaScript programs in a database, and for restoring and running the programs without recompiling them. Be aware that these two methods have not been released in Release 6.10, but will be in 6.20 (see sidebar on page 29).

A Checklist of Techniques Learned

Once again, to review what you have learned in this section, **Figure 7** summarizes the key techniques for accessing JavaScript from your ABAP programs.

Figure 7 *Accessing JavaScript from ABAP*

Technique	ABAP or JavaScript Means
Create a JavaScript runtime context	Create an instance of CL_JAVA_SCRIPT by calling the factory method CREATE.
Compile, execute, and destroy JavaScript programs within a JavaScript context	Use the methods COMPILE, EXECUTE, EVALUATE, and DESTROY.

(continued on next page)

Figure 7 *(continued)*

Technique	ABAP or JavaScript Means
Error handling	Check attributes LAST_CONDITION_CODE and LAST_ERROR_MESSAGE.
Access ABAP objects from within JavaScript	Use the methods BIND and BIND_INSTANCE.
Access ABAP structures and substructures	Bind structures to JavaScript objects with properties that correspond to the ABAP structure.
Access internal tables	Bind tables to a corresponding JavaScript array object. Use the attribute “length” and the method “append”.
Call ABAP methods	Use the method “invoke”.
Upon method invocation, pass values of basic JavaScript type by reference	Use JavaScript objects of type SAPAbapMethParam.
Catch classical ABAP exceptions	Use “invoke” to invoke the ABAP method and pass the predefined parameter EXCEPTION.
Destroy a JavaScript context and undo all data bindings	Use the method ABAP_DESTRUCTOR or CLEAR the CL_JAVA_SCRIPT instance.

✔ ***For More Information…***

The online documentation for the ABAP Workbench is an excellent source of information about the CL_JAVA_SCRIPT class.

When to Use JavaScript vs. ABAP

So, now you have some new choices for web-enabling your SAP applications. Let’s explore when a JavaScript BSP is the right option, when calling JavaScript from ABAP might be a better choice, and when to avoid JavaScript altogether.

When you’re creating BSPs, JavaScript is always a good choice. This is especially true if you are working with people such as web page designers that don’t understand script fragments written in ABAP. JavaScript is an industry-standard tool that they probably already know, which makes it easier for them to work with or improve your page layout. However, if you have designers with ABAP knowledge, ABAP is obviously the better choice for direct access to SAP components.

Of course, you should also consider the task at hand when choosing the most suitable language. Some things are simply easier to accomplish in JavaScript. For example, processing two tables in parallel is laborious in ABAP, while in JavaScript you can implement it quite easily with a simple loop statement.

 ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.

A Sneak Preview of Release 6.20

Release 6.20 will further enhance the SAP Web Application Server, in particular with more JavaScript features. Here is a preview of some key enhancements:

- ✓ Support for data references (REF TO DATA) when binding objects.
- ✓ Support for access to static attributes and the invocation of static methods when binding objects.
- ✓ Support of structures with more than 256 components when binding objects.
- ✓ Stable versions of the FREEZE and THAW methods in the CL_JAVA_SCRIPT class.

Additional development is planned, although full details weren't available at publication time.

However, avoid using JavaScript in BSPs in the following situations:

- When you need to bind either many ABAP objects, or very large ABAP objects,[18] because accessing them from JavaScript is quite slow
- If you need to process ABAP objects, such as hashed and sorted tables, that aren't accessible from JavaScript

Remember that you don't need to build a web application with ABAP or JavaScript scripting exclusively. BSPs based on JavaScript and ABAP can coexist peacefully in your application. Instead, use the guidelines in this section to determine whether you really need to choose a single scripting language.

Using JavaScript within ABAP programs — that is, using the class `CL_JAVA_SCRIPT` — is always advisable if you want to exploit JavaScript features that make some tasks, such as processing strings, easier. Another reason to use JavaScript is if you need to reuse JavaScript code that was written externally. Just keep in mind that you don't have to use external tools. The JavaScript engine is only a few method calls away.

The situations in which you should not use JavaScript in your ABAP programs are the same as for BSPs. In particular, binding many objects or large objects is not a good idea.

Conclusion

In Release 6.10 of the SAP Web AS, the SAP kernel successfully integrates an interpreter (Mozilla, JavaScript 1.5) for programming languages other than ABAP for the first time. With the introduction of JavaScript interpreter support, you can now use JavaScript in Business Server Pages to develop web applications. Plus, you can also access the JavaScript interpreter from any ABAP program. The ability to bind ABAP objects to JavaScript provides direct access to the SAP world, and thus to a wide variety of powerful features.

As a result, many SAP customers are finding that building web applications on SAP servers using BSPs is a welcome leap forward. Developing BSPs

[18] For example, tables with thousands of lines.

is rather easy, and offers significantly reduced implementation time over other technologies. One customer, Hotelplan (a Swiss travel business) even built their entire online booking system[19] with BSPs and JavaScript, based on the SAP FI (Financials) and CO (Controlling) modules. Moreover, the bottom line is that application maintenance and enhancement simply costs less. It's not surprising that several customers we know of are re-implementing their existing web applications using BSPs.

[19] Comprising about 20 pages.

Now it's time for you to decide for yourself. We hope the information in this article encourages you to try these techniques, and see why it's easier than ever to web-enable your SAP applications.

Acknowledgements

We would like to thank Klaus Ziegler for his contributions to this article.

Peter Januschke studied mathematics at the Technische Universität Karlsruhe, Germany. At the Institute of Applied Mathematics, he researched computer arithmetic, verification numerics, and programming languages for Scientific Computation. In addition, he participated in three European research projects, focusing on compiler technology. After earning a doctorate, Peter joined the Business Programming Languages Group at SAP AG in 1998. Peter is responsible for the ABAP scanner and syntax check, the ABAP extended program check, arithmetic and conversion, and the BSP compiler. He can be reached at peter.januschke@sap.com.

Holger Janz is a software developer at SAP AG in the Business Programming Languages Group. Prior to joining SAP in 1997, he studied computer science at the University of Rostock and Constance, where he focused on object-oriented programming languages. As a member of the development team at SAP, Holger's responsibilities include parts of the ABAP Virtual Machine and the ABAP Compiler, and the integration of the JavaScript Virtual Machine into the SAP Web Application Server. His team is also responsible for the integration of Java into the SAP Web Application Server architecture. He can be reached at holger.janz@sap.com.

 ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.