

## Paper 32-26

## JavaScript Tutorial by Example

Iza Peszek, Merck & Co., Inc., Rahway, NJ

### ABSTRACT

This paper presents a code walk-thru for a useful JavaScript application designed to construct a call for a SAS ® macro. It is intended to help you learn the basics of JavaScript by example. It is not a comprehensive JavaScript tutorial: you are encouraged to consult books on JavaScript to read more about the presented techniques. Some good references are given at the end of the paper.

You can use the code “as is” without understanding HTML tags, but the paper will be easier to follow if you have some familiarity with HTML syntax.

Note that the presented application is designed in such a way that SAS programmers can generate it for any macro in their library using a relatively simple (guess what?) SAS macro!

### INTRODUCTION

The purpose of the application, which will determine its design, is to:

- display the parameters of a specific SAS macro, as well as the information about the macro and about its parameters,
- allow users to enter the values for the macro parameters,
- perform a validation of user input,
- construct a valid macro call.

Since we want to gather user input, we are going to use FORM FIELDS. This approach will let us provide some default values for macro parameters or restrict the values to a finite set of specific choices. User input can be processed and validated on a client by JavaScript.

We also want to display information about the macro and about its parameters. Thus, we need to separate the display area into **information area** and **data input area**. The easiest way to do so is to use a FRAMESET containing a Top Frame (information window) and a Bottom Frame (parameter setting window).

It is logical to display the information about the macro at the start of the application, and the information about an individual parameter at the time the user is working with this parameter – i.e., when the cursor is inside the parameter’s field.

This roughly defines a draft of the application structure. Next we will take care of the implementation details. Figure 1 presents a snapshot of the application window.

### FRAMESET

The application consists of three HTML documents: parent frameset, top frame, and bottom frame.

Let us start with the parent document. It defines a frameset with two frames, named “topframe” and “bottomframe”. By naming the frames, we will be able to refer to them from within our JavaScript code. The frameset reserves 25% of the display area for the top

frame and suppresses the borders, so the displayed document does not look fragmented. We also define the source documents for the frames.

```
<html>
<head>
<title>
  macro call maker for dohtml
</title>
<script language="JavaScript">

  window.moveTo(0,0);
  window.outerWidth=screen.availWidth;
  window.outerHeight=screen.availHeight;

</script>
</head>
<frameset rows="25%,*"
  framespacing="0" border="0"
  frameborder="0">
<frame name="topframe"
  src="dohtml_top.htm">
<frame name="bottomframe"
  src="dohtml_form.htm"
  scrolling="auto">
<noframes>
This page uses frames but your browser
does not support them
</noframes>
</frameset>
</html>
```

The first three lines of the JavaScript code maximize the size of the window before it is displayed. This code works only in Netscape Communicator.

The <NOFRAMES>...<NOFRAMES> tag is ignored by browsers which understand frames. Older browser, which can not deal with frames, will display the message specified between these tags, so the users can understand why the application is not working.

### TOP FRAME

The document displayed in a top frame is very simple:

```
<html>
<body>
<p><b>Macro <a href="dohtml.htm"
target="_parent"> %dohtml:</a></b>
The macro to produce interactive
graphs for web display (continue
description...)
</body>
</html>
```

Note that the hyperlink uses a keyword *target*, so the hyperlinked document will display in a “\_parent” window, i.e. in a frameset document.

### BOTTOM FRAME

The bottom frame is the meat of the application. Let us start with a FORM to allow entering values for the macro parameters. The FORM is placed inside a TABLE, so the

fields are aligned nicely:

```
<body>
<form name="dohtml" >
<table >
<tr><td align="right">device =</td>
<td>
<select size="1" name="Rdevice"
onFocus="showhelp(1 )"
onBlur="hidehelp()" >
<option value="gif733 ">gif733
</option>
<option value="gif570 ">gif570
</option>
</select></td>
<td rowspan="3">
<p>
<input type="button"
value="Default Values"
name="xxxxxxxxx2" onClick=
"parent.window.location.reload();
return true">
</p>
<p>
<input type="button" name="xxxxxxxxx3"
value="Macro Info"
onClick="showInfo()" ></p>
<p>
<input type="button" name="xxxxxxxxx1"
value="Create Call"
onClick="makeCall()" >
</td></tr>
<tr><td align="right">plotname =</td>
<td>
<input type="text" name="Rplotname"
size=" 30"
value="c:\theplot.gif"
onFocus="showhelp(2 )"
onBlur="hidehelp()" >
</td></tr>
<tr><td align="right">datain =</td>
<td>
<input type="text" name="Rdatain"
size=" 30" value="plotdata "
onFocus="showhelp(3 )"
onBlur="hidehelp()" >
</td>
</tr>
</table>
</form>
</body>
```

Let us look at the form closely.

The form is enclosed in a pair of tags: `<form>...</form>`. The opening tag gives the form a name – so we can refer to it in JavaScript code. Inside a form, we will place form fields. There are several types of form fields: text boxes, text areas, drop-down lists, radio buttons, checkboxes, etc. We will be using only text boxes and drop-down lists.

### TEXTBOXES

The textbox has a syntax

```
<input type="text" name= , size=,
value=, onFocus=, onBlur= >
```

(there are more optional modifiers, but we will not need them).

By specifying a NAME for a form field, we can easily refer to it in JavaScript code. The keyword SIZE determines the length of a textbox. Each browser renders the length slightly differently, so you need to find the best size by trial and error. In most cases, the size of 20 or 30 works well.

The keyword VALUE specifies the text displayed in a field. Finally, the two event handlers - ONFOCUS and ONBLUR - are used to specify JavaScript functions to run when user enters a field (e.g., by clicking a mouse on a field) or leaves the field (e.g., by pressing "TAB" or clicking the mouse somewhere outside the field).

We will talk about JavaScript functions later.

### DROP-DOWN LISTS

The drop-down lists are constructed using a pair of tags `<select></select>`, inside which are one or more `<option>` tags, defining elements in the list.

The `<select>` tag has several modifiers. The modifier NAME will let us refer to the drop-down list in the code. The modifier SIZE specifies how many elements are displayed to the user when the list is collapsed. The ONFOCUS and ONBLUR event handlers define the action triggered by the respective events.

The `<option>` tag resembles the anchor (`<a>`) tag: the displayed value is specified between `<option>` and `</option>` tags; while the "passed" value is specified by the modifier "VALUE". If you wish, you can also use a modifier SELECTED to specify which value is selected by default:

```
<option value="gif733 " selected>
gif733 </option>
```

Remember that only one of the `<option>` tags should have this modifier.

### BUTTONS

Most – but not all – forms also use BUTTON fields. A button field is used to perform some action when the button is clicked. There are three types of buttons: SUBMIT, RESET and BUTTON. The SUBMIT button, when clicked, will send form data for processing – unless the field modifiers redefine the onClick action. The RESET button clears all fields. The regular button does not do anything when clicked unless you specifically write an action code.

The buttons we use in our application are all regular buttons. As for all form elements, we specify a name for the button (keyword NAME) as well as the text displayed on its face (keyword VALUE). Finally, we specify a JavaScript function to be executed when the button is clicked (event handler onClick).

### TABLE

The form fields are placed in a table, which lets us align them nicely. The simple syntax for a table is as follows:

```
<table>
<tr>
```

```

<td>...</td>
<td>...</td>
...
</tr>
<tr>
<td>...</td>
<td>...</td>
...
</tr>
...
</table>

```

The tags `<tr>` and `</tr>` mark the beginning and end of a row; the tags `<td>` and `</td>` mark the beginning and the end of a column. The tags can have several modifiers. In our example, we use a modifier `ALIGN` to specify the alignment of text in a cell. We also use a modifier `ROWSPAN` to merge several cells vertically (a modifier `COLSPAN` can be used to merge cells horizontally).

Having defined our input fields, it's time to add some action!

#### “DEFAULT VALUES” BUTTON

Let us start with a simplest action: resetting everything to default values. Our “Default Values” button has an `onClick` action defined as

```

onClick=
"parent.window.location.reload()"

```

The action consists of two JavaScript statements, separated with semicolon. Alternatively, we could specify the name of a JavaScript function which performs these actions. This is useful for complicated actions consisting of many steps. We will see an example of this scenario later.

A statement

```
parent.window.location.reload()
```

acts as if user performed a “hard reload” of a page (shift/Reload). It will display the document in its initial form – as if opened for the first time – discarding any changes made by user.

#### “MACRO INFO” BUTTON

When “Macro Info” button is clicked, it calls a JavaScript function `showInfo()`. Do not forget a set of brackets at the end – JavaScript needs them to understand that you mean a function, not a variable.

A function `showInfo()`, as well as the rest of JavaScript code used on this page, is defined inside a `HEAD` tag:

```

<HEAD>
<SCRIPT language="JavaScript">
function showInfo(){
parent.topframe.location='dohtml_top.htm';
}
</SCRIPT>
</HEAD>

```

In theory, JavaScript code can be placed anywhere on a page. By placing it in the `HEAD` tag, however, we avoid a common pitfall of amateurish scripts: we are making sure that the script is fully loaded before the page is displayed. If we placed it inside a `BODY` tag, user could have performed an action calling some JavaScript function before the browser loaded the definition of this function. This would have caused a JavaScript error!

Our `showInfo()` function does not take any parameters and has just one line of body:

```

parent.topframe.location=
'dohtml_top.htm'

```

Recall that our top frame was named “topframe”, so the function requests that the top frame displays the document “dohtml\_top.htm”. What about the word “parent” in front? Well – remember that we are working inside the document “bottomframe”, which knows nothing about any other documents! But it knows about its parent – the frameset – and the frameset knows its “children”: “topframe” and “bottomframe”. If you work in a frameset, and you want to refer to one frame from another frame, you need to work your way “up” through a document which recognizes both frames!

#### FUNCTIONS `SHOWHELP()` AND `HIDEHELP()`

Let us take a look at some other functions – `showHelp()` and `hideHelp()` – which are called from each of our input form fields:

```

<input type="text" name="Rplotname"
size=" 30" value="c:\theplot.gif"
onFocus="showhelp(2)"
onBlur="hidehelp()" >

```

When a cursor is placed in a field, the `onFocus` event is fired and a function `showHelp()` is called. This function takes one parameter – in this case its value is 2. When a cursor leaves a field, the `onBlur` event is fired, and a function `hideHelp()` is called.

The `showHelp()` function is defined as follows:

```

<head>
<script language="JavaScript">
function showInfo(){..}
function showhelp(num)
{
thetext =
"<html><head></head><body>";
thetext += "<table><tr><td>";
thetext += helptext[num];
thetext += "</td></tr></table>";
thetext += "</body><html>";
parent.topframe.document.open();

parent.topframe.document.write(thetext
);
parent.topframe.document.close();
}
</script>
</head>

```

The first five lines define a text string *thetext*, containing

a proper HTML document. Note the operator "+=", used to append the string on the right hand side to the string variable on the left hand side. The string *thetext* is hard-coded, except for an entry *helptext[num]* – which is an element of an array *helptext* defined outside of this function. The array *helptext* has 3 elements, corresponding to our 3 input fields. It contains “tool-tip help” for the fields and is defined as follows:

```
<head>
<script language="JavaScript">
var helptext=new Array();
helptext[1]=
'graphic device- select from list';
helptext[2]=
'name of the clickable plot';
helptext[3]=
'data set used to produce clickable
plot, with additional variables: `';
helptext[3]+='<br>url_var, url_desc,
xpixels and ypixels `';

function showInfo(){..}
function showhelp(num){..}
</script>
</head>
```

The first line,

```
var helptext=new Array()
```

is a constructor for an array *helptext*. The following lines define the elements of the array – the indexes have to be enclosed in square brackets.

Note: Variables defined inside a definition of a function have a local scope: they are visible only to this function. Variables defined outside any function have global scope: they are visible to all JavaScript on a page. Thus, the variable *thetext* exists only inside the function *showHelp()*, while the array *helptext* is visible to all JavaScript on a page.

Argument passed to *showHelp()* function identifies which element of the array should be used inside the string *thetext*.

The last three lines of function *showHelp()* display the constructed HTML document in the top frame. They open the top frame document for input, write text, and close the input stream so the control is returned to browser.

A function *hideHelp()* is a simplified, complementary version of *showHelp()* – it displays an empty document in the top frame, clearing any field-specific information.

```
function hidehelp(){
thetext = "<html><head></head><body>
</body><html>";
parent.topframe.document.open();
parent.topframe.document.write(thetext
);
parent.topframe.document.close();
}
```

## MAKECALL() FUNCTION

Our main button – “Create Call” calls a function *makeCall()*:

```
function makeCall(){
//definition here
}
```

The purpose of this function is to gather and validate the values supplied by user. If the validation fails, the function will display an error message and will not produce a macro call until the error is corrected. Once validation passes, the function constructs the macro call, and displays the call in separate browser window.

This function is defined in a generic fashion, so its code does not need to be modified when we change the fields in our form. In fact, the whole application is designed so that only the input fields on the form, macro information in the top frame document, and the tool tips corresponding to the form fields need to be changed if we want to create a call-maker for another SAS macro.

To facilitate this generic behavior, we are using a special naming convention for our fields: the fields name is constructed from the name of the macro parameter, with a prefix R (required keyword parameter), K (non-required keyword parameter) or P (positional parameter). Soon we will see how this naming convention helps us to make our function generic.

In the beginning, we declare some variables used inside the function:

```
var theval;
var callText;
var currentObj;
```

Although it is not strictly necessary, declaring the variables up front improves readability of a program.

We also start defining the value of a string variable *callText*:

```
callText = "<h1> Macro call for macro
";
callText += document.forms[0].name;
callText += "</h1><p><pre>";
callText +=
"%"+document.forms[0].name;
callText += "<br>";
```

In particular, we are using a property *document.forms[0].name* – the name of our form. The collection *forms* contains all the forms on a page. In JavaScript, the enumeration starts at 0, so *forms[0]* denotes the first form on a page.

Why don't we use the name "dohtml" directly? The reason is that we want our function to be generic, so we do not need to modify it for another SAS macro.

As we go along, we'll keep appending to *callText* string until our macro call document is constructed in full.

Next, we are going to process each of our fields:

```

For (i=0;
i<document.dohtml.elements.length;
i++)
{
// loop-processing
}

```

This is a simple iteration over all elements of the form. The collection

"document.forms[0].elements" contains all elements belonging to the first form on a page; the property "length" is a total number of form elements.

Inside the "for" loop, we refer to the form elements as "document.forms[0].elements[i]":

```

// loop-processing
// code part 1
currentObj =
document.forms[0].elements[i];
if (currentObj.type != "button") {
  if (currentObj.type == "text"){
    theval = trim(currentObj.value);
  }
  else{
    theval=
    trim(currentObj.options[currentObj.selectedIndex].value);
  }
// continue loop processing
// code part 2
}

```

This snippet assigns the current form field to the variable *currentObj* (so we can have shorter, more manageable reference to the current form element) and determines the value of the field.

The first "if-else" statement tests the field and determines whether the field is a button or an input field. Note that JavaScript requires "==" to test for equality; an "=" operator is an assignment operator. For all fields except the drop-down list, we can get the value using *currentObj.value* property. For list boxes, the method is slightly different: we figure out the value corresponding to the selected index. Thus, we need a second "if-else" statement, which determines the type of the field. Note that we trim the value to get rid of potential leading or trailing blanks. There is no trim() function in JavaScript, so we need to write our own and place it in somewhere between <SCRIPT> and </SCRIPT> tags:

```

function trim(thestring){
  var newstring=thestring;
  for (j=0; j< thestring.length;
j++){
    if (newstring.substr(0,1) == " ")
  ){
    newstring=newstring.substr(1);
  }
  if
(newstring.substr(newstring.length-
1,1) == " "){ newstring =
newstring.substr(0,newstring.length-
1);}
}
return newstring;
}

```

Next, we perform the basic validation of the fields.

```

// code part 2
if (theval == ""){
  if (currentObj.name.substr(0,1)=="P"
||
currentObj.name.substr(0,1)=="R"){
    alert("The parameter
"+currentObj.name.substr(1)+ " is
required. \r\n Please supply the
value." );
    currentObj.focus();
    return false;
  }
}
// continue loop processing
// code part 3

```

First of all, if the field is empty, we need to check whether it is required. Positional parameters and required keyword parameters can not have empty values, so our first "if-else" statement checks for parameter type (recall our naming convention) by looking at the first character of the field's name:

```
currentObj.name.substr(0,1).
```

The substr() method takes two arguments: starting position (0) and the length of the produced substring (1). Note again that the enumeration starts with 0.

If the parameter is required and the field is empty, JavaScript will display a modal message – built-in function alert() – asking user to correct the error. It also places the cursor in the problematic field (*surrentObj.focus()*). Finally, it "returns" so no other statements are processed.

Once user corrects the problem and clicks "Create call" button, the function is called again to call the *makeCall()* function after the error is corrected.

If the value of a field is not empty, but the parameter is a keyword parameter, we append some blanks to *callText* to make the display look nicer:

```

// code part 3
else {
  if (currentObj.name.substr(0,1)=="K"
||currentObj.name.substr(0,1)=="R"
){
    for (j=0;
j<(9-currentObj.name.length);
j++){
      callText+=" ";
    }
  }
// continue loop processing
// code part 4

```

This way, the "=" signs will be aligned in the created macro call:

```

%dohtml(
  device = gif733,
  plotname = c:\theplot.gif,

```

```
datain = plotdata)
```

For keyword parameters, we need to include the name of the parameter in the macro call. Once again our naming convention comes in handy: we can get the name of the parameter from the name of the field – we just need to delete the prefix. The `substr(1)` method takes care of this. By omitting a second parameter of the `substr()` method, we ask for all characters of the field name, starting with the second. Then, we append the “=” sign:

```
// code part 4
if (currentObj.name.substr(0,1)=="K"
||
    currentObj.name.substr(0,1)=="R"){
    callText+=
        currentObj.name.substr(1) + " =
";
}
// continue loop processing
// code part 5
```

Next, we append the value of the field and insert a comma and a soft line break. The closing curly brackets close our “if-else” and “for” loops.

```
// code part 5
callText+= theval+",<br>";
}
}
// loop done
```

Next, we complete the definition of `callText` string by deleting the “,<BR>” at the end and appending the closing parenthesis and closing HTML tags:

```
// loop done - construct call text
CallText =
callText.substr(0,callText.length-5);
callText+=")<br></pre>";
```

Finally, if there was no validation errors, we open a separate window and write a `callText` string to display the macro call.

```
// display call text
var newWin = window.open('',
'MacroCall', 'height=550,width=500');
newWin.document.write(callText);
newWin.document.close();
newWin.focus();
```

The `window.open()` method returns a window object which we assign to the variable `newWin`, so we can refer to it later. This method takes several arguments: a URL for the document to place in a new window, a title for a new window, and the list of windows modifiers. In our case, we specified the height and width of a new window in pixels.

Once the new window is opened, we write the constructed macro call and close the input stream to return control to the browser. Finally, we call the `focus()` method to display the created window.

## CONCLUSION

This completes the analysis of a JavaScript used in our application.

Our field validation was limited to checking if all required fields have been specified. You can use more involved validation techniques by writing your own validation functions. For example, you can define a function `exampleValidation()`, taking as an argument a form field element:

```
function exampleValidation(element){
if ( element.value.substr(0,1) == "c")
{
    alert(element.name.substr(1) + " can
not start with 'c!'");
    element.focus();
    return false;
}

else {return true}
}
```

Then, you would place a call to this function before generic validation takes place:

```
function makeCall(){
if
(ExampleValidation(document.forms[0].R
plotname) == false){return false;}
...
}
```

You can also validate the form fields based on the value of other fields:

```
Function exampleValidation2(element1,
element2){
If (element1.value == "GIF533" &&
    Element2.value == "details")
{
    alert("You can not request details
for small display");
    element.focus();
    return false;
}

function makeCall(){
if (ExampleValidation2(
document.forms[0].Rdevice,
document.forms[0].Kdetails) == false){
    return false;
}
...
}
```

By making JavaScript function quite generic, we can place them in a separate document and include this external code in our HTML files as follows:

```
<HEAD>
<SCRIPT language="JavaScript"
source="../scripts/mysoure.js">
</SCRIPT>
```

## REFERENCES

Goodman D. "JavaScript Bible", IDG Books, 1998

Wyke R.A., Gilliam J.D., and Ting C. "Pure JavaScript", SAMS, 1999

Peszek I., Troxell J. "Interactive SAS Macro Catalogs". Proceedings of PharmaSUG'2000

## TRADEMARK INFORMATION

SAS is a registered trademark or a trademark of SAS Institute Inc. in the USA and other countries. © indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

## CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Iza Peszek  
 Merck & Co., Inc  
 P.O. Box 2000, RY33-404  
 Rahway, NJ, 07065-0900  
 Work Phone: 732-594-3623  
 Fax: 732-594-6075  
 Email: [izabella\\_peszek@merck.com](mailto:izabella_peszek@merck.com)

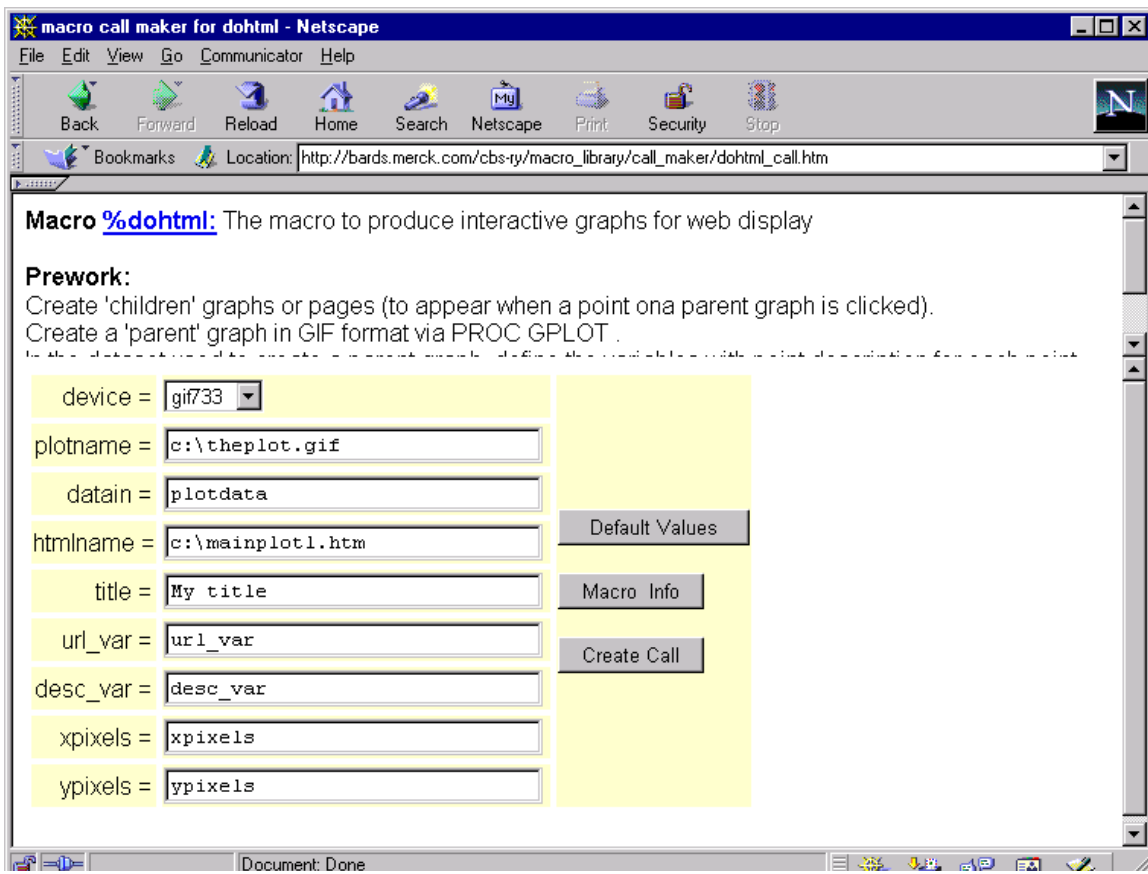


Figure 1 : A snapshot of the application screen