
JavaScript Tips for ASP.NET – Part 1

Summary: This article provides useful JavaScript tips and tricks for ASP.NET developers. (14 printed pages)

Contents

1. Get DOM elements with one simple method.....	1
2. Add/Remove event handlers.....	2
3. View the current HTML state of your page.....	4
4. Disable a button after click to prevent subsequent postbacks to server.....	5
5. Use references to deeply nested objects.....	6
6. Create your own watermark textbox.....	6
7. Center your window on the screen.....	7
8. Validate max character length in a text area.....	10
9. Display confirmation before the page is closed.....	11
10. Format number.....	12
11. Swap table rows on the client-side.....	14

Overview

There is no doubt that ASP.NET server controls make your life easy on server-side. Third-party components like **Karamasoft UISuite™** make it a lot easier on both server-side and client-side. However, you may still need some knowledge on client-side programming to provide your website visitors with a powerful user interface.

JavaScript is the most commonly used language for client-side programming regardless of your server-side development environment. Even though this article may be applied to other languages and frameworks, it is more appropriate for ASP.NET developers because most of the tips are accompanied with sample code in ASPX and C#.

1. Get DOM elements with one simple method

There is a cross-browser **getElementById** method of the document object, which returns a reference to the object with the specified value of the ID attribute. Prototype JavaScript Framework (<http://www.prototypejs.org>) provides a convenient JavaScript method with a short name to return DOM elements:

```
function $() {  
    var elements = new Array();
```

```
for (var i = 0; i < arguments.length; i++) {  
    var element = arguments[i];  
    if (typeof element == 'string')  
        element = document.getElementById(element);  
    if (arguments.length == 1)  
        return element;  
    elements.push(element);  
}  
return elements;  
}
```

One nice thing about this method is that you can pass either one argument or multiple arguments. If one argument is passed it returns a reference to the object with the specified ID value. Otherwise it returns an array of references to the objects with the specified ID values. Another nice thing about this method is that it takes objects as well as strings as arguments.

Instead of the following:

```
var elem = document.getElementById('TextBox1');
```

You can now call the **\$** method as follows:

```
var elem = $('TextBox1');
```

Or you can return multiple DOM elements with one simple method call:

```
var elemArr = $('TextBox1', 'TextBox2');
```

2. Add/Remove event handlers

You can easily bind/unbind a function to an event so that it gets called whenever the event fires on the object.

Internet Explorer DOM (Document Object Model) provides **attachEvent** and **detachEvent** methods, while Mozilla DOM provides **addEventListener** and **removeEventListener** methods to add/remove event handlers.

You can use the following cross-browser JavaScript methods to add/remove your event handlers:

```
function AddEventHandler(obj, eventName, functionNotify) {
    if (obj.attachEvent) {
        obj.attachEvent('on' + eventName, functionNotify);
    }
    else if (obj.addEventListener) {
        obj.addEventListener(eventName, functionNotify, true);
    }
    else {
        obj['on' + eventName] = functionNotify;
    }
}

function RemoveEventHandler(obj, eventName, functionNotify) {
    if (obj.detachEvent) {
        obj.detachEvent('on' + eventName, functionNotify);
    }
    else if (obj.removeEventListener) {
        obj.removeEventListener(eventName, functionNotify, true);
    }
    else {
        obj['on' + eventName] = null;
    }
}
```

You can call the above methods as follows:

JavaScript

```
function AddKeyDownEventHandler(obj) {
    AddEventHandler(obj, 'keydown', KeyDownEventHandler);
}

function KeyDownEventHandler(evt) {
    alert('Event key code: ' + GetEventKeyCode(evt));
}

function GetEventKeyCode(evt) {
    return evt.keyCode ? evt.keyCode : evt.charCode ? evt.charCode :
    evt.which ? evt.which : void 0;
}

function BodyOnloadHandler(evt) {
    AddKeyDownEventHandler(document.getElementById('<%=txtKeyDown.Client
ID%>'));
}
```

ASPX

```
<body onload="BodyOnloadHandler()">
<asp:TextBox ID="txtKeyDown" runat="server"
CssClass="TextBox"></asp:TextBox>
```

3. View the current HTML state of your page

When you do **View Source** in your browser, it displays the state of your page when the page is loaded. However, when you have dynamic content in your page, especially in AJAX environment, you might need to see the current source of your page.

Type the following into your address bar and press enter.

Internet Explorer

```
javascript:'<xmp>'+window.document.body.outerHTML+'</xmp>'
```

Firefox

```
javascript: '<xmp>' + document.getElementsByTagName('html')[0].innerHTML + '  
</xmp>'
```

Or you can combine these two into one JavaScript statement to make it cross-browser:

```
javascript: '<xmp>' + ((document.all) ? window.document.body.outerHTML :  
document.getElementsByTagName('html')[0].innerHTML) + '</xmp>'
```

4. Disable a button after click to prevent subsequent postbacks to server

As a developer you may never double-click a submit button, but your website users might do so, and execute the submit button code twice. They might as well click multiple times while it's still in progress. If you're running database operations or sending emails on the button click handler, double-clicks may create unwanted results in your system. Even if you're not doing anything critical, it will overload your server with unnecessary calls.

To prevent subsequent postbacks to server, as soon as the user clicks the submit button you can disable it and change its text in the onclick event handler as follows:

JavaScript

```
function DisableButton(buttonElem) {  
    buttonElem.value = 'Please Wait...';  
    buttonElem.disabled = true;  
}
```

ASPX

```
<asp:button id="btnSubmit" runat="server" Text="Submit" />
```

C#

```
protected void Page_Load(object sender, EventArgs e)  
{
```

```
    btnSubmit.Attributes.Add("onclick", "DisableButton(this);" +  
Page.ClientScript.GetPostBackEventReference(this,  
btnSubmit.ID.ToString()));  
}
```

5. Use references to deeply nested objects

JavaScript objects can be nested using the **dot** operator as the following:

```
tableElem.rows[0].cells[0]
```

If you do multiple operations on the above construct, it is better to define a variable to reference it because each dot operator causes an operation to retrieve that property. For example, if you need to process table cells inside a for loop, define a local variable to reference the cells collection and use it instead as follows:

```
var cellsColl = tableElem.rows[0].cells;  
for (var i = 0; i < cellsColl.length; i++) {  
    cellsColl[i].style.backgroundColor = '#FF0000';  
}
```

Even though the above code references the **cells** collection through the local variable, it still checks the length property of the cells collection in each step of the loop. Therefore, it is better to store **cellsColl.length** property in a local variable by getting its value once as follows:

```
for (var i = 0, loopCnt = cellsColl.length; i < loopCnt; i++)
```

6. Create your own watermark textbox

The main purpose of a watermark is to provide information to the user about the textbox without cluttering up the page. You have probably seen many examples of this in **search textboxes** in websites. When a watermarked textbox is empty, it displays a message to the user. Once the user types some text into the textbox, the watermarked text disappears. When the user leaves the textbox the watermarked text appears again if the content of the textbox is empty.

You can easily change your textbox to provide watermark behavior by adding **onfocus** and **onblur** event handlers. In the focus event, clear the textbox if its text matches the watermark text. In the blur event, set the text back to watermark text if the textbox empty.

JavaScript

```
function WatermarkFocus(txtElem, strWatermark) {  
    if (txtElem.value == strWatermark) txtElem.value = '';  
}  
  
function WatermarkBlur(txtElem, strWatermark) {  
    if (txtElem.value == '') txtElem.value = strWatermark;  
}
```

ASPX

```
<asp:TextBox ID="txtWatermark" runat="server" />
```

C#

```
protected void Page_Load(object sender, EventArgs e)  
{  
    string strWatermark = "Search Karamasoft.com";  
    txtWatermark.Text = strWatermark;  
    txtWatermark.Attributes.Add("onfocus", "WatermarkFocus(this, '" +  
strWatermark + "')");  
    txtWatermark.Attributes.Add("onblur", "WatermarkBlur(this, '" +  
strWatermark + "')");  
}
```

7. Center your window on the screen

You can center your windows opened with the **window.open** method on the screen. Both Internet Explorer and Mozilla DOM provide window screen object that has **availWidth** and **availHeight** properties to retrieves the width and height of the working area of the system's screen. All you need to do is to get the difference

between the screen width (or height) value and the window width (or height) value and divide it by 2.

Since you might want to make this functionality reusable, you can create a wrapper method to call `window.open` method by replacing its `features` argument that includes attributes such as width and height. You can parse width and height attributes in the `features` argument value, and find the appropriate left and top values to center the window and append left and top attributes to the `features` argument value.

The **features** argument usually looks like the following:

```
'width=400,height=300,location=no,menubar=no,resizable=no,scrollbars=no, status=yes,toolbars=no'
```

First, we need to create a method to parse the width and height values in the `features` string. Since we might want to use this method for other purposes to split name/value pairs such as parsing query strings to find query string values, let's make it a generic method.

```
function GetAttributeValue(attribList, attribName, firstDelim,
secondDelim) {
    var attribNameLowerCase = attribName.toLowerCase();
    if (attribList) {
        var attribArr = attribList.split(firstDelim);
        for (var i = 0, loopCnt = attribArr.length; i < loopCnt; i++) {
            var nameValueArr = attribArr[i].split(secondDelim);
            for (var j = 0, loopCnt2 = nameValueArr.length; j < loopCnt2;
j++) {
                if (nameValueArr[0].toLowerCase().replace(/\s/g, '') ==
attribNameLowerCase && loopCnt2 > 1) {
                    return nameValueArr[1];
                }
            }
        }
    }
}
```

This method takes **three arguments**: a name/value pair list, the attribute name to retrieve its value, the first delimiter and the second delimiter. The first delimiter will be comma and the second delimiter will be equal sign in this case. The first delimiter would be ampersand and the second delimiter would be equal sign to parse query string variables.

Then define the methods to retrieve available screen width and height values.

```
function GetScreenWidth() {  
    return window.screen.availWidth;  
}  
  
function GetScreenHeight() {  
    return window.screen.availHeight;  
}
```

We can now create our **window.open** wrapper method to center the window by using these methods.

```
function WindowOpenHelper(sURL, sName, sFeatures, centerWindow) {  
    var windowLeft = '';  
    var windowTop = '';  
    if (centerWindow) {  
        var windowHeight = GetAttributeValue(sFeatures, 'width', ',',  
            '=');  
        windowLeft = (typeof(windowWidth) != 'undefined') ? ',left=' +  
            ((GetScreenWidth() - windowHeight) / 2) : '';  
  
        var windowHeight = GetAttributeValue(sFeatures, 'height', ',',  
            '=');  
        windowTop = (typeof(windowHeight) != 'undefined') ? ',top=' +  
            ((GetScreenHeight() - windowHeight) / 2) : '';  
    }  
    window.open(sURL, sName, sFeatures + windowLeft + windowTop);  
}
```

This method takes **four arguments**: URL of the document to display, the name of the window, a list of feature items, and a Boolean to indicate whether the window should be centered.

You can now call **WindowOpenHelper** method in your page.

JavaScript

```
function OpenWindowCentered(windowWidth, windowHeight) {  
    WindowOpenHelper('http://www.karamasoft.com', 'WindowCentered',  
'width=400,height=300,location=no,menubar=no,resizable=no,scrollbars=no,  
,status=yes,toolbars=no', true);  
}
```

ASPX

```
<asp:LinkButton ID="lbOpenWindowCentered" runat="server"  
OnClick="OpenWindowCentered(); return false;">Open window  
centered</asp:LinkButton>
```

8. Validate max character length in a text area

HTML input type of text element provides a built-in **MaxLength** property to set the maximum number of characters that the user can enter. However, the **TextArea** element does not have such property to limit the number of characters that can be entered. When you have an ASP.NET TextBox control with `TextMode="MultiLine"` in your web page, it is rendered as an HTML TextArea element and you cannot use `MaxLength` property to set the maximum number characters.

What you can do is to define a **keypress** event handler for the TextBox control to check the length of the text inside the text area and cancel the event if the `MaxLength` is reached.

JavaScript

```
function ValidateMaxLength(evnt, str, maxLength) {  
    var evntKeyCode = GetEventKeyCode(evnt);  
  
    // Ignore keys such as Delete, Backspace, Shift, Ctrl, Alt, Insert,  
Delete, Home, End, Page Up, Page Down and arrow keys
```

```
var escChars = ",8,17,18,19,33,34,35,36,37,38,39,40,45,46,";
if (escChars.indexOf(',') + evntKeyCode + ',' == -1) {
    if (str.length >= maxLength) {
        alert("You cannot enter more than " + maxLength + "
characters.");
        return false;
    }
}
return true;
}
```

ASPX

```
<asp:TextBox ID="txtValidateMaxLength" runat="server"
TextMode="MultiLine" />
```

C#

```
protected void Page_Load(object sender, EventArgs e)
{
    txtValidateMaxLength.Attributes.Add("onkeypress", "return
ValidateMaxLength((window.event) ? window.event : arguments[0],
this.value, 5)");
}
```

9. Display confirmation before the page is closed

If you need to display a confirmation before the user closes the window, you can use the **beforeunload** event of the window object to display a confirmation message.

JavaScript

```
function AddUnloadHandler() {
    AddEventHandler(window, 'beforeunload', HandleUnload);
}
```

```
function HandleUnload() {  
    var strConfirm = 'Please make sure you saved your changes before  
closing the page.';  
    if (document.all) {  
        window.event.returnValue = strConfirm;  
    }  
    else {  
        alert(strConfirm);  
        var evnt = arguments[0];  
        evnt.stopPropagation();  
        evnt.preventDefault();  
    }  
}
```

ASPX

```
<body onload="AddUnloadHandler()">
```

10. Format number

You can use the following method to format your numbers or currency values. It takes **four arguments**: the number to format, the number of decimal places, a Boolean to indicate whether zeros should be appended, and a Boolean to indicate whether commas should be inserted to separate thousands.

```
function FormatNumber(num, decimalPlaces, appendZeros, insertCommas) {  
    var powerOfTen = Math.pow(10, decimalPlaces);  
    var num = Math.round(num * powerOfTen) / powerOfTen;  
    if (!appendZeros && !insertCommas) {  
        return num;  
    }  
    else {  
        var strNum = num.toString();  
        var posDecimal = strNum.indexOf(".");
```

```
    if (appendZeros) {
        var zeroToAppendCnt = 0;
        if (posDecimal < 0) {
            strNum += ".";
            zeroToAppendCnt = decimalPlaces;
        }
        else {
            zeroToAppendCnt = decimalPlaces - (strNum.length -
posDecimal - 1);
        }
        for (var i = 0; i < zeroToAppendCnt; i++) {
            strNum += "0";
        }
    }
    if (insertCommas && (Math.abs(num) >= 1000)) {
        var i = strNum.indexOf(".");
        if (i < 0) {
            i = strNum.length;
        }
        i -= 3;
        while (i >= 1) {
            strNum = strNum.substring(0, i) + ',' +
strNum.substring(i, strNum.length);
            i -= 3;
        }
    }
    return strNum;
}
}
```

11. Swap table rows on the client-side

You can swap table rows on the client-side by swapping the cell contents of the current rows and the row to swap. The following method takes **three parameters**: the current table row element, a Boolean to indicate whether the row should move up, and a Boolean to indicate whether the first row should be ignored.

```
function SwapRows(rowElem, dirUp, ignoreFirstRow) {
    var rowElemToSwap = (dirUp) ? rowElem.previousSibling :
rowElem.nextSibling;

    // Firefox returns a blank text node for the sibling
    while (rowElemToSwap && rowElemToSwap.nodeType != 1) {
        rowElemToSwap = (dirUp) ? rowElemToSwap.previousSibling :
rowElemToSwap.nextSibling;
    }

    if (rowElemToSwap && !(ignoreFirstRow && rowElemToSwap.rowIndex ==
0)) {
        var rowCells = rowElem.cells;
        var colInner;
        for (var i = 0, loopCnt = rowCells.length; i < loopCnt; i++) {
            colInner = rowCells[i].innerHTML;
            rowCells[i].innerHTML = rowElemToSwap.cells[i].innerHTML;
            rowElemToSwap.cells[i].innerHTML = colInner;
        }
    }
}
```