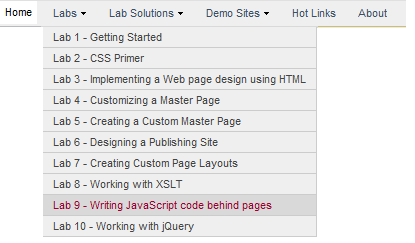
## Writing JavaScript Code behind Pages

**Lab Time**: 45 minutes

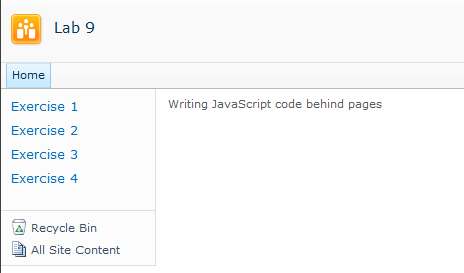
### Exercise 1: Adding JavaScript to a page in a SharePoint site

In this exercise you will work through the basic steps of adding JavaScript to code within a page in a SharePoint site. After getting a simple event handler to execute on a button click event, you will also work through the process of modifying your code to register event handlers dynamically. In the final steps of this exercise, you will move the JavaScript code out of the ASPX page and into a dedicated JavaScript file.

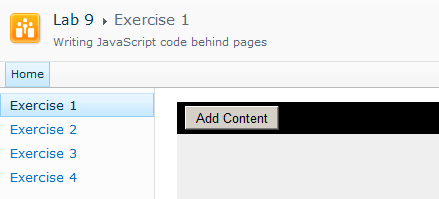
1. In the browser, navigate to the Brand Camp labs site collection at **http://[[COLLAB-SITE]]** and navigate to the child site for **Lab 9** by using the global navigation menu of the top-level site.



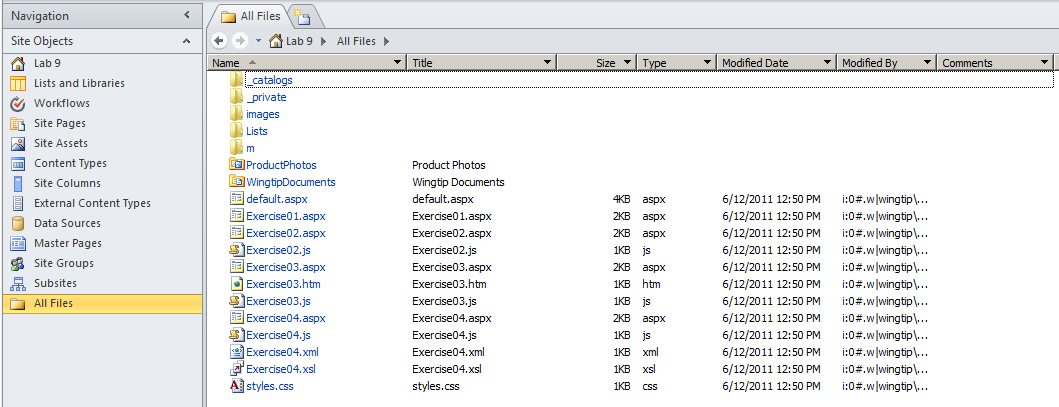
1. You should now be at the home page in the **Lab 9** site. You should see four links in the Quick Launch Bar to navigate to the pages you will be working on through the exercises in this lab.



1. In the browser, navigate to the page for **Exercise 1** by clicking the navigation link in the Quick launch. As you can see, the page contains a command button that currently has no behavior behind it. Your main task in Exercise 1 is to write and register an event handler function that executes when the user clicks the button.



1. Launch SharePoint Designer 2010 and open the site at **http://[[COLLAB-SITE]]/Lab09**. If you are prompted to log in, enter the site collection owner as **[[AD\_DOMAIN]]\[[USERNAME]]**. For example, if you were supplied the user account login credentials for Ken Sanchez with a user account name **ken** and a domain name **SBO,** then theaccount name for logging in should be entered as **SBO\ken**. The password for all accounts should be set to **Password1**.
2. Once the site has opened in SharePoint Designer, click the **All Files** link at the bottom of the **Site Objects** section. You should be able to see the files for pages named **Exercise01.aspx**, **Exercise02.aspx**, **Exercise03.aspx** and **Exercise04.aspx**. You should also see that there is an associated JavaScript file for each of these pages with the exception of **Exercise01.aspx**.



1. At this point, you should have the **Lab 9** site open in both the browser and in SharePoint Designer. Leave both of these open as you will be moving back and forth between them for all the remaining exercises of this lab.
2. In SharePoint Designer, open the page named **Exercise01.aspx** in advanced edit mode. You should see the page contains placeholders for **PlaceHolderAdditionalPageHead** and **PlaceHolderMain**. You should also see that there is HTML content inside **PlaceHolderMain** that defines an input element of type **button** with an **id** of **cmdAddContent** as well as an empty div element with an **id** of **Div1**.

<asp:Content ContentPlaceHolderId="PlaceHolderAdditionalPageHead" runat="server">

<link rel="stylesheet" type="text/css" href="styles.css" />

</asp:Content>

<asp:Content ContentPlaceHolderId="PlaceHolderMain" runat="server">

<div id="MainContentDiv" >

<div id="Toolbar" >

<input id="cmdAddContent" type="button" value="Add Content" />

</div>

<div id="ContentAreaDiv" >

<div id="Div1" />

</div>

</div>

</asp:Content>

1. Now it is time to add some JavaScript code to the page. Begin by adding a script block to **PlaceHolderAdditionalPageHead.**

<asp:Content ContentPlaceHolderId="PlaceHolderAdditionalPageHead" runat="server">

<link rel="stylesheet" type="text/css" href="styles.css" />

<script type="text/javascript">

</script>

</asp:Content>

1. Now add a JavaScript function named **myCommandHandler** inside the script block that dynamically adds content into **Div1**. You can add content to **Div1** using the string value shown in the following example or come up with your own string that is more clever and/or humorous.

<script type="text/javascript">

function myCommandHandler() {

var div1 = $get('Div1');

div1.innerHTML = "Writing JavaScript is fun!";

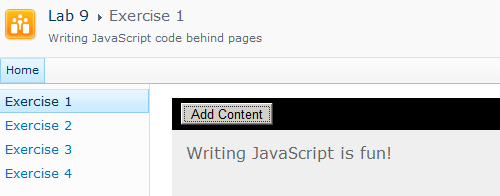
}

</script>

1. Now wire up the event handler by adding the **onclick** attribute to the input element for the button and giving the attribute a value of **myCommandHandler** followed by parenthesis.

<input id="cmdAddContent" type="button" value="Add Content" onclick="myCommandHandler()" />

1. Save your work by saving the page in the SharePoint Designer. Now return to the browser and refresh the page. Click the command button and you should see your message written in the page contents. Note that to test your work you must first refresh the page to ensure that the JavaScript you wrote is downloaded into the browser.



While wiring an event handler up in this fashion using an HTML element attribute such as **onclick** works, it is not recommended. There are better ways to register event handlers to achieve higher levels of reuse and maintainability. You will now modify the page with a page load event handler that registers the event handler for the button to accomplish the same goal.

1. In SharePoint Designer, modify **Exercise01.aspx** by removing the **onclick** attribute from the input element. Save your changes.

<input id="cmdAddContent" type="button" value="Add Content" />

1. In the browser, refresh the page and test clicking the button. The button should not have any behavior behind it now since you removed the **onclick** attribute.
2. In SharePoint Designer, modify **Exercise01.aspx** to register the event handler a different way. Accomplish this by adding a function named **myPageLoad** which registers the event handler named **myCommandHandler** for the **click** event of **cmdButton1** using **$addHandler**. Also, use the **Sys.Application.add\_load** function to register the **myPageLoad** function so that it executes as soon as the document has been loaded.

<asp:Content ContentPlaceHolderId="PlaceHolderAdditionalPageHead" runat="server">

<link rel="stylesheet" type="text/css" href="styles.css" />

<script type="text/javascript">

Sys.Application.add\_load(myPageLoad);

function myPageLoad() {

var cmdButton1 = $get('cmdAddContent');

$addHandler(cmdButton1, 'click', myCommandHandler);

}

function myCommandHandler() {

var div1 = $get('Div1');

div1.innerHTML = "Writing JavaScript is fun";

}

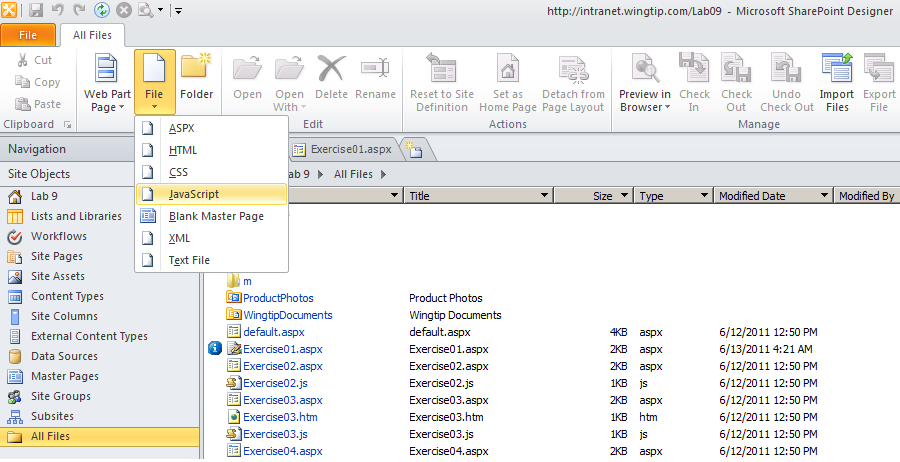
</script>

</asp:Content>

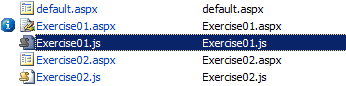
1. In the browser, refresh the page and test clicking the button. The button should now execute code that adds content to **Div1** just as it did before. The difference is that now you have done it without wiring up event handlers using HTML element attributes. This is a much better approach to wiring up event handlers because it adheres to the principles of *Unobstructive JavaScript*.

There is one final improvement you will now make to the JavaScript code you have written in Exercise 1. More specifically, you will move your JavaScript code out of the page named **Exercise01.aspx** and add it to a dedicated JavaScript file named **Exercise01.js**. You will then add a script link to **Exercise01.aspx** to link to the JavaScript code **Exercise01.js**.

1. In SharePoint Designer, click the **All Files** link at the bottom of the **Site Objects** section. You should be able to see the files at the root of the site on the right-hand side of the screen. Drop down the **File** menu located in the **New** group in the Ribbon and click the **JavaScript** menu command to create a new JavaScript file at the root of the site.



1. Once the JavaScript file has been created, rename it to **Exercise01.js**.



1. Open **Exercise01.js** so you can edit it. At this point, the file should be empty.
2. Now, using the Windows clipboard, cut all the JavaScript code out of the script block in **Exercise01.aspx** and paste that code into **Exercise01.js**. The content of **Exercise01.js** should now look like this:

Sys.Application.add\_load(myPageLoad);

function myPageLoad() {

var cmdButton1 = $get('cmdAddContent');

$addHandler(cmdButton1, 'click', myCommandHandler);

}

function myCommandHandler() {

var div1 = $get('Div1');

div1.innerHTML = "Writing JavaScript is fun";

}

1. Save your changes to **Exercise01.js**.
2. Finally, modify the placeholder named **PlaceHolderAdditionalPageHead** in **Exercise01.aspx** to link to the JavaScript code in **Exercise01.js**.

<asp:Content ContentPlaceHolderId="PlaceHolderAdditionalPageHead" runat="server">

<link rel="stylesheet" type="text/css" href="styles.css" />

<script src="Exercise01.js" type="text/javascript" ></script>

</asp:Content>

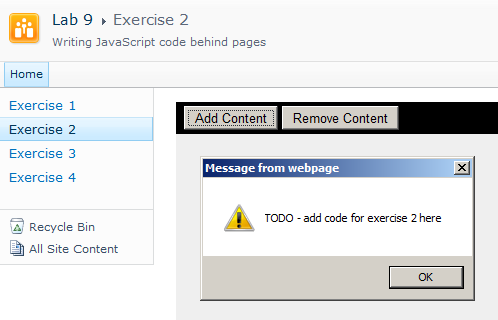
1. Save your changes to **Exercise01.aspx**.
2. Return to the browser and refresh **Exercise01.aspx**. Test your work by clicking the button. The button should now execute code that adds text-based content to **Div1** just as it did before. The difference, however, is that now you accomplished the same goal while maintaining your JavaScript code in a separate, dedicated JavaScript file.

You have now been through the steps of creating and linking to a dedicated JavaScript file. Note that you will not be required to create or link to other JavaScript files in the remaining exercises of this lab. For your convenience (*and to make things go faster*), the next three exercises provide you with JavaScript files that have already been created and linked.

### Exercise 2: Using JavaScript to dynamically create HTML elements

In this exercise you will write event handlers that dynamically add and remove HTML elements to and from a page.

1. In the browser, navigate to the page for **Exercise 2** by clicking the navigation link in the Quick launch. The page contains two command buttons. Click each of the buttons to see what they do. You should observe that there are already event handlers wired to both of these button that display simple dialogs.



1. In SharePoint Designer, open **Exercise02.aspx** in advanced edit mode. Examine the contents of **PlaceHolderMain**. You should see that there are two input elements of type button with **id** values of **cmdAddContent** and **cmdRemoveContent**. There is also an empty div element below with an **id** of **ContentAreaDiv**. This is the element into which you will write dynamic content.

<asp:Content ContentPlaceHolderId="PlaceHolderMain" runat="server">

<div id="MainContentDiv" >

<div id="Toolbar" >

<input id="cmdAddContent" type="button" value="Add Content" />

<input id="cmdRemoveContent" type="button" value="Remove Content" />

</div>

<div id="ContentAreaDiv" />

</div>

</asp:Content>

1. Examine what’s inside **PlaceHolderAdditionalPageHead**. You can see there is a script link to an associated JavaScript file named **Exercise02.js**.

<asp:Content ContentPlaceHolderId="PlaceHolderAdditionalPageHead" runat="server">

<link rel="stylesheet" type="text/css" href="styles.css" />

<script src="Exercise02.js" type="text/javascript" ></script>

</asp:Content>

1. Now you can close **Exercise02.aspx**. You will not be required to make any changes to this file in this exercise. All required changes will be made to **Exercise02.js**.
2. Open **Exercise02.js** and examine the JavaScript code that is already inside this file. At the top, there is a 4 element array with string values that has been assigned to a variable named **beatles**. There is also a call to **Sys.Application.add\_load** which registers a function named **myPageLoad** to execute once the page has loaded. When **myPageLoad** executes, it registers two event handler functions named **addContentHandler** and **removeContentHandler** so that they execute when the user clicks the buttons on the page.

var beatles = Array("John", "Paul", "George", "Ringo");

Sys.Application.add\_load(myPageLoad);

function myPageLoad() {

$addHandler($get('cmdAddContent'), 'click', addContentHandler);

$addHandler($get('cmdRemoveContent'), 'click', removeContentHandler);

}

function addContentHandler() {

alert('TODO - add code for exercise 2 here');

}

function removeContentHandler() {

alert('TODO - add code for exercise 2 here');

}

1. Begin by removing the call to **alert** in **addContentHandler**. Next, use the **$get** function to obtain a reference to the div element with an **id** of **ContentAreaDiv**. Assign this reference to a new variable named **ContentAreaDiv**.

function addContentHandler() {

var ContentAreaDiv = $get("ContentAreaDiv");

}

1. Next, dynamically create a new unordered list element (**ul**) by calling **document.createElement** and assigning the return value to a new variable named **list1**. Next, add this new **ul** element to **ContentAreaDiv** by calling the **appendChild** method.

function addContentHandler() {

var ContentAreaDiv = $get("ContentAreaDiv");

var list1 = document.createElement("ul");

ContentAreaDiv.appendChild(list1);

}

1. Now it's time to dynamically create the items that will be added to the list. Add a few blank lines in between the call to **createElement** and the call to **appendChild**. Next, add a **for** loop which enumerates through the **beatles** array and creates list item elements (**li**) and appends them to the list. You will be required to call **document.createTextNode** using the text value from the array and to call the **appendChild** method to properly append the text node to the list item. The following listing shows the code that is required to accomplish this.

function addContentHandler() {

var ContentAreaDiv = $get("ContentAreaDiv");

var list1 = document.createElement("ul");

for (var index = 0; index < beatles.length; index++) {

var listItem = document.createElement("li");

var textNode = document.createTextNode(beatles[index]);

listItem.appendChild(textNode);

list1.appendChild(listItem);

}

ContentAreaDiv.appendChild(list1);

}

1. Save your changes to **Exercise02.js**.
2. Return to the browser and refresh **Exercise02.aspx**. Click the **Add Content** button. Your page should dynamically display a new unordered list with the contents of the array.



1. Now see what happens if you click the **Add Content** button more than once. Things do not seem correct because it adds a new instance of the unordered list each time you click the button, without removing the list or lists that were already there. Your next task is to write the code to dynamically remove list elements when they are no longer needed.



1. Return to SharePoint Designer so you can continue to edit **Exercise02.js**. Begin by removing the call to **alert** in **removeContentHandler**. Next, write a **while** loop that removes all the child element nodes from the div elements with the **id** of **ContentAreaDiv**. You can accomplish this with the following code.

function removeContentHandler() {

var ContentAreaDiv = $get("ContentAreaDiv");

while (ContentAreaDiv.childNodes.length > 0) {

ContentAreaDiv.removeChild(ContentAreaDiv.childNodes[0]);

}

}

1. Modify the **addContentHandler** function by adding a call to **removeContentHandler** as the first line. This will ensure that your code always removes all existing child elements from the target div element before adding any new one.

function addContentHandler() {

removeContentHandler();

var ContentAreaDiv = $get("ContentAreaDiv");

var list1 = document.createElement("ul");

for (var index = 0; index < beatles.length; index++) {

var listItem = document.createElement("li");

var textNode = document.createTextNode(beatles[index]);

listItem.appendChild(textNode);

list1.appendChild(listItem);

}

ContentAreaDiv.appendChild(list1);

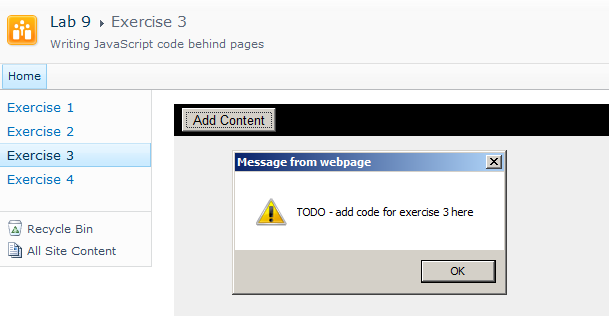
}

1. Return to the browser and refresh page **Exercise02.aspx**. Click the **Add Content** button and then the **Remove Content** button. Your page should dynamically display a new unordered list and then remove it. Also you should be able to click the **Add Content** button more than once yet not see more than one instance of the list.

### Exercise 3: Retrieving content from across the network with AJAX requests

In this exercise you will write the JavaScript code to perform an AJAX style request retrieving remote content from the Web server across the network.

1. In the browser, navigate to the page for **Exercise 3** by clicking the navigation link in the Quick launch. The page contains a command button. Click it to see what it does. You should observe that there is already an event handler wired to it that displays a simple dialog.



1. In SharePoint Designer, open **Exercise03.aspx** in advanced edit mode. You should see that there is a script link in **PlaceHolderAdditionalPageHead** that links to **Exercise03.js**.

<asp:Content ContentPlaceHolderId="PlaceHolderAdditionalPageHead" runat="server">

<link rel="stylesheet" type="text/css" href="styles.css" />

<script src="Exercise03.js" type="text/javascript" ></script>

</asp:Content>

1. Examine the contents of **PlaceHolderMain**. There is an **input** element of type **button** with **id** value of **cmdAddContent**. There is also an empty div element below with an id of **Div1**. This is the element into which you will write your dynamic content.

<asp:Content ContentPlaceHolderId="PlaceHolderMain" runat="server">

<div id="MainContentDiv" >

<div id="Toolbar" >

<input id="cmdAddContent" type="button" value="Add Content" />

</div>

<div id="ContentAreaDiv" >

<div id="Div1" />

</div>

</div>

</asp:Content>

1. Open **Exercise03.js**. You can see that there is a command handler function named **addContentHandler** that has already been wired up to the command button **cmdAddContent**.

Sys.Application.add\_load(myPageLoad);

function myPageLoad() {

var cmdButton1 = $get('cmdAddContent');

$addHandler(cmdButton1, 'click', addContentHandler);

}

function addContentHandler() {

alert('TODO - add code for exercise 3 here');

}

1. Your current task is to modify the implementation of **cmdAddContent**. Begin by removing the line that calls the **alert** function. Next, add the JavaScript code required to make an AJAX-style **GET** request using a **WebRequest** object. This retrieves the content from the file named **Exercise03.htm** which is located inside the same folder as **Exercise03.aspx**. Before calling the **invoke** method on the **WebRequest** object, make sure to configure it to make a callback to a new function named **myCallbackMethod**. The callback method named **myCallbackMethod** should include a parameter named **response**. The JavaScript code you write for this step should be written to match the following code listing.

function addContentHandler() {

var request = new Sys.Net.WebRequest();

request.set\_url("Exercise03.htm");

request.set\_httpVerb("GET");

request.add\_completed(myCallbackMethod);

request.invoke();

}

function myCallbackMethod(response) {

// TO DO: add impementation

}

1. Write an implementation for the function **myCallbackMethod** that retrieves the content from the AJAX request by calling the **get\_responseData** method on the **response** parameter. Next, write the content into the div element with an **id** of **Div1**.

function myCallbackMethod(response) {

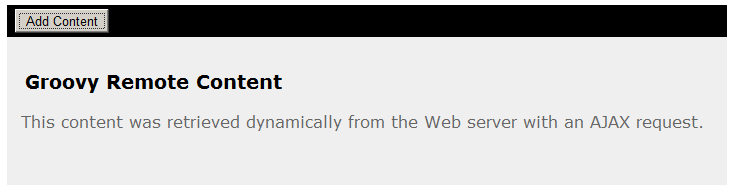
var ResponseText = response.get\_responseData();

var div1 = $get('Div1');

div1.innerHTML = ResponseText;

}

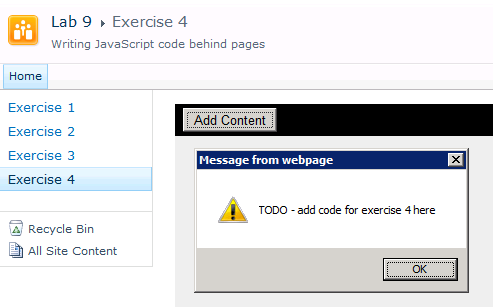
1. Now it is time to test your work. Save your changes to **Exercise03.js** in the SharePoint Designer. Return to the browser and refresh page **Exercise03.aspx**. Click the **Add Content** button. The JavaScript code you have written should retrieve the contents of **Exercise03.htm** and display it on the page without requiring a postback of the hosting page.



### Exercise 4: Performing a Client-side XSLT Transform

In this exercise you will write the JavaScript code to perform an AJAX style request retrieving and XML document and an XSLT file. You will then add the code required to perform a client-side XSL transform to produce HTML from the XML document for display in the browser.

1. In the browser, navigate to the page for **Exercise 4** by clicking the navigation link in the Quick launch. The page contains a command button. Click it to see what it does. You should observe that there is already an event handler wired to it that displays a simple dialog.



1. In SharePoint Designer, open **Exercise04.aspx** in advanced edit mode. You should see that there is a script link in **PlaceHolderAdditionalPageHead** that links to **Exercise04.js**.

<asp:Content ContentPlaceHolderId="PlaceHolderAdditionalPageHead" runat="server">

<link rel="stylesheet" type="text/css" href="styles.css" />

<script src="Exercise04.js" type="text/javascript" ></script>

</asp:Content>

1. Examine the contents of **PlaceHolderMain**. There is an **input** element of type **button** with **id** value of **cmdAddContent**. There is also an empty div element below with an id of **Div1**. This is the element into which you will write your dynamic content after you perform the XSLT transform.

<asp:Content ContentPlaceHolderId="PlaceHolderMain" runat="server">

<div id="MainContentDiv" >

<div id="Toolbar" >

<input id="cmdAddContent" type="button" value="Add Content" />

</div>

<div id="ContentAreaDiv" >

<div id="Div1" />

</div>

</div>

</asp:Content>

1. Open **Exercise04.js** in advanced edit mode. You can see that there is a command handler function named **addContentHandler** that has already been wired up to **cmdAddContent**.

Sys.Application.add\_load(myPageLoad);

function myPageLoad() {

var cmdButton1 = $get('cmdAddContent');

$addHandler(cmdButton1, 'click', addContentHandler);

}

function addContentHandler() {

alert('TODO - add code for exercise 4 here');

}

1. In this exercise, you will create a new JavaScript object that will be used to keep track of various pieces of information about each asynchronous request. Start by creating a top-level variable named **WingtipRequest** just above the **addContentHandler** method. Create a new JavaScript object and assign it to the variable.

var WingtipRequest = new Object();

function addContentHandler() {

alert('TODO - add code for exercise 4 here');

}

1. Next, modify the implementation of **cmdAddContent** by removing the call to the **alert** function. Next, add a line that accesses the object held by the **WingtipRequest** variable and assigns a value of **false** to a property named **processed**. Add two more properties named **xsl** and **xml** and initialize them to a value of **null**. These two properties will be used to store the XSL code and the XML document used in the XSL transform.

var WingtipRequest = new Object();

function addContentHandler() {

WingtipRequest.processed = false;

WingtipRequest.xsl = null;

WingtipRequest.xml = null;

}

1. Add the code to execute two AJAX-style requests to retrieve two files named **Exercise04.xsl** and **Exercise04.xml** from across the network using **WebRequest** objects. The code you need to write for this step will be almost identical to the code you wrote in the previous exercise to retrieve an HTML document. Create a callback method to process **Exercise04.xsl** named **xlsLoadCompleted**. Create a second callback method to process **Exercise04.xml** named **xmlLoadCompleted**.

var WingtipRequest = new Object();

function myCommandHandler() {

WingtipRequest.processed = false;

var xslRequest = new Sys.Net.WebRequest();

xslRequest.set\_url("Exercise04.xsl");

xslRequest.set\_httpVerb("GET");

xslRequest.add\_completed(xlsLoadCompleted);

xslRequest.invoke();

var xmlRequest = new Sys.Net.WebRequest();

xmlRequest.set\_url("Exercise04.xml");

xmlRequest.set\_httpVerb("GET");

xmlRequest.add\_completed(xmlLoadCompleted);

xmlRequest.invoke();

}

function xlsLoadCompleted(response) {

// TO DO: implement function

}

function xmlLoadCompleted(response) {

// TO DO: implement function

}

1. Implement **xlsLoadCompleted** to retrieve the XML content of **Exercise04.xsl** by calling the **get\_xml** method of the **response** parameter. Store the XML content in a dynamically created property of the **WingtipRequest** object named **xsl**.

function xlsLoadCompleted(response) {

WingtipRequest.xsl = response.get\_xml();

}

1. Implement **xmlLoadCompleted** to retrieve the XML content of **Exercise04.xml** by calling the **get\_xml** method of the **response** parameter. Store the XML content in a dynamically created property of the **WingtipRequest** object named **xml**.

function xmlLoadCompleted(response) {

WingtipRequest.xml = response.get\_xml();

}

1. Add a function named **ProcessXSLT**. Call it from **xslLoadCompleted** and **xmlLoadCompleted.**

function xlsLoadCompleted(response) {

WingtipRequest.xsl = response.get\_xml();

ProcessXSLT();

}

function xmlLoadCompleted(response) {

WingtipRequest.xml = response.get\_xml();

ProcessXSLT();

}

function ProcessXSLT() {

}

1. Note that the **ProcessXSLT** will be called in scenarios before both files have been retrieved. Therefore, conduct a check to ensure that the **xls** property and the **xml** property of the **WingtipRequest** object are not **null**. If either property contains a **null** value, return from the **ProcessXSLT** to avoid attempting the transform. Do the same type of check to ensure that the **processed** property of the **WingtipRequest** object does not have a value of **true**.

function ProcessXSLT() {

if (WingtipRequest.xsl === null) return;

if (WingtipRequest.xml === null) return;

if (WingtipRequest.processed === true) return;

}

1. Create an if statement to determine whether the browser is Internet Explorer or not.

function ProcessXSLT() {

if (WingtipRequest.xsl === null) return;

if (WingtipRequest.xml === null) return;

if (WingtipRequest.processed === true) return;

if (Sys.Browser.agent === Sys.Browser.InternetExplorer) {

// add implementation for Internet Explorer

}

else {

// add implementation for other browsers

}

}

1. If the browser is Internet Explorer, process the XSL transform by calling the **transformNode** method on the **xml** property of the **WingtipRequest** object. When calling the **transformNode** method, pass the **xsl** property of the **WingtipRequest** object as a parameter. Assign the return value from the call to **transformNode** to the **innerHTML** property of the div with an **id** of **Div1**.

if (Sys.Browser.agent === Sys.Browser.InternetExplorer) {

var ie\_output = WingtipRequest.xml.transformNode(WingtipRequest.xsl);

$get("Div1").innerHTML = ie\_output;

}

1. The way in which you process XSL transforms for browsers other than Internet Explorer is quite different. Begin by creating a variable named **Div1** and initializing it with a reference to the div element with an **id** of **Div1**. Next, write a **while** loop that deletes any child nodes that exist in **Div1**.

if (Sys.Browser.agent === Sys.Browser.InternetExplorer) {

var ie\_output = WingtipRequest.xml.transformNode(WingtipRequest.xsl);

$get("Div1").innerHTML = ie\_output;

}

else {

var Div1 = $get("Div1");

// delete any existing child nodes

while (Div1.childNodes.length > 0) {

WingtipRequest.element.removeChild(WingtipRequest.element.childNodes[0]);

}

}

1. Next, create a new JavaScript object by calling new on the JavaScript constructor method named **XSLTProcessor**. Assign this object to a new variable named **xsltProcessor**. Call the **importStylesheet** method on **xsltProcessor** passing the **WingtipRequest.xsl** property as a **xsltProcessor**. Next, call the **transformToFragment** method on **xsltProcessor** passing both **WingtipRequest.xsl** and theJavaScript **document** object as parameters. Take the return value of the **transformToFragment** method and assign it to a new variable named **non\_ie\_output**. Finally, add the nodes created by the transform to the div element **Div1** by calling **appendChild**.

if (Sys.Browser.agent === Sys.Browser.InternetExplorer) {

var ie\_output = WingtipRequest.xml.transformNode(WingtipRequest.xsl);

$get("Div1").innerHTML = ie\_output;

}

else {

var Div1 = $get("Div1");

while (Div1.childNodes.length > 0) {

WingtipRequest.element.removeChild(WingtipRequest.element.childNodes[0]);

}

var xsltProcessor = new XSLTProcessor();

xsltProcessor.importStylesheet(WingtipRequest.xsl);

var non\_ie\_output = xsltProcessor.transformToFragment(WingtipRequest.xml, document);

Div1.appendChild(non\_ie\_output);

}

1. The last step is to assign a value of true to the **WingtipRequest.processed** property. When you are done, your implementation of **ProcessXSLT** should match the following code listing.

function ProcessXSLT() {

if (WingtipRequest.xsl == null) return;

if (WingtipRequest.xml == null) return;

if (WingtipRequest.processed === true) return;

if (Sys.Browser.agent === Sys.Browser.InternetExplorer) {

var ie\_output = WingtipRequest.xml.transformNode(WingtipRequest.xsl);

$get("Div1").innerHTML = ie\_output;

}

else {

var Div1 = $get("Div1");

while (Div1.childNodes.length > 0) {

WingtipRequest.element.removeChild(WingtipRequest.element.childNodes[0]);

}

var xsltProcessor = new XSLTProcessor();

xsltProcessor.importStylesheet(WingtipRequest.xsl);

var non\_ie\_output = xsltProcessor.transformToFragment(WingtipRequest.xml, document);

Div1.appendChild(non\_ie\_output);

}

WingtipRequest.processed = true;

}

1. Now it is time to test your work. Save your changes to **Exercise04.js** in the SharePoint Designer. Return to the browser and refresh page **Exercise04.aspx**. Click the **Add Content** button. The JavaScript you have written should retrieve the contents of the XML document and the XSLT file and then apply the transform and display its output on the page.

