

# Internet Mark-up Languages - CO32036

## AJAX – Asynchronous Javascript And XML

mjr 4.10.06

### ***Lecture Contents***

- Some basic JavaScript.
- How to use alert boxes for debugging.
- How to read a data file without changing web page.
- How to change text in a web page without refreshing the page
- How a web page can pull data out of an XML file and display it.

### ***AJAX - Definition***

The term AJAX stands for “Asynchronous JavaScript And XML”. In practice, the technique can be used and is used without the need for any XML technology, but we’re going to teach that anyway.

AJAX includes a way of updating information on a web page without updating the page. This leads into a very useful way of running applications within browsers (like Outlook OWA). The implication is that it is now possible to write applications specially to be downloaded and run within browser windows. These could be games, but business applications like word processors and graphics editors are also possible. Google already offers a browser-based word processor (Writely) and a graphics editor (Picasa)

Ajax is not a programming language - most of its programming works on the client side using JavaScript. Nor is AJAX a server technology - it runs on ordinary web servers, downloading files from the server in the usual way. In some AJAX applications there may be PHP or some other server application running, but that is not at the heart of what AJAX is about.

Having said what JavaScript is not, let us now try and say what it is: AJAX is a browser technology, which uses only JavaScript in the browser page. It employs a very efficient way of updating information, DOM the Document Object Model (which is a standard part of the JavaScript language). It is also a way of making applications smaller, faster, easier to use.

...and then there’s the portability. Don’t forget the portability. AJAX will run in most of the mainstream browsers in use today.

## ***AJAX and Standards***

Some technologies are based on proprietary standards. A good example is Flash: before you can use Flash you have to buy the authoring tool. AJAX is based on open standards. Anyone can sit at an ordinary computer and author AJAX applications in an ordinary text editor. (Nevertheless, tools can be very useful!)

AJAX is based on the following international standards:

- JavaScript (ECMAScript)
- XML
- HTML
- CSS
- HTTP

Although Netscape originated the JavaScript language, Microsoft quickly responded by releasing a non-compatible dialect, called JScript. Users then appealed to the European Computer Manufacturers Association to provide a negotiated compromise, which was then published as ECMAScript. Even today, some functions in JavaScript are only practicable by writing two sets of code - one for Microsoft Internet Explorer and one for the rest of us.

## ***How AJAX Works***

AJAX uses two basic tricks. These are:

- HTML request to server: XMLHttpRequest()
- The Document Object Model (DOM)  
(We'll be looking at DOM in greater depth in another lecture)

In typical operation, the user triggers an HTML event, such as `onClick` or `onmouseover`. A JavaScript program then sends an HTTP request to the server. The server supplies a file, as servers usually do. That file may be XML, HTML, text or any other data format. The JavaScript in the current browser page selects part of the data in the file for display. Then a JavaScript statement alters the browser display according to the data in the document. This usually means displaying some new data at an appropriate point on the web page.

## ***Programming Examples***

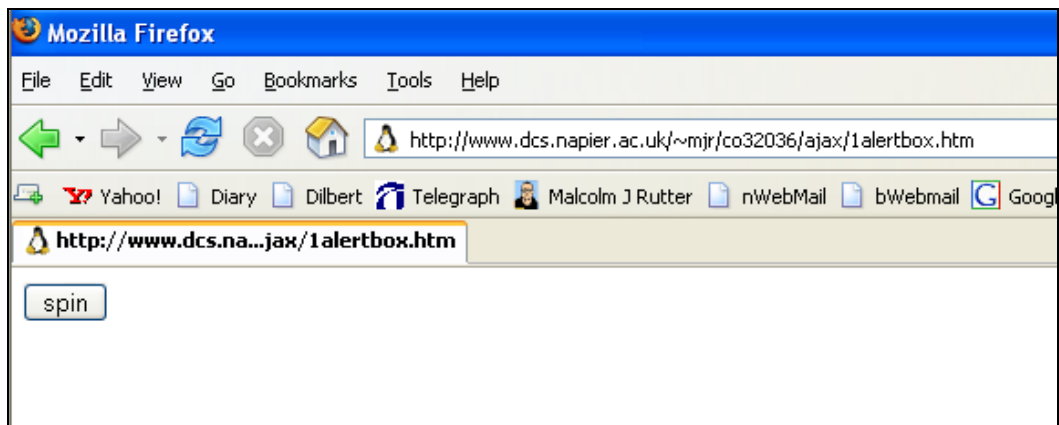
If you are impatient, you can learn about AJAX very quickly by simply looking up how to use XMLHttpRequest() in both standard and Microsoft technologies, and then learning DOM and HTML events. That's where we're going, but we're going to take a gentler route:

## 1. alertbox.htm

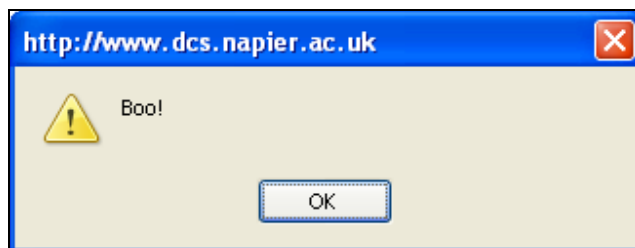
The alert box is probably the easiest piece of JavaScript to get to work. Having used it to gain confidence, it remains a very powerful debugging tool:

```
<html>
<head> </head>

<body>
<input type='button' onclick="javascript:alert('Boo!')"
value='spin' />
</body>
</html>
```



Most HTML entities (like links, images and form elements) have permissible behaviours associated with them. In this example, a button is made to perform a JavaScript instruction when it is clicked. That instruction causes an alert box to pop up.



## 2. abfunc.htm

Although JavaScript may be used in-line as in the previous simple example, it is better practice to put the JavaScript into the head of the page as a group of functions. This may be left there during debugging, but when it works, the page is made even tidier by making the JavaScript a separate file which is then included in the page at download time.

```
<html> <head>
<script type='text/javascript'>
function popup(message)
    { alert(message); }
</script> </head>

<body>
<input type='button' onmouseover="popup('Boo!')"
value='wash' />
</body> </html>
```

The browser screen and alert box look just the same as in the previous example.

## IE Only

(3iesuck.htm)

The next program gets a text file from the server and displays the contents in an alert box. To do this, it uses the instruction, “XMLHttpRequest()”. To keep it simple, I’ve written a program that will only work in Internet Explorer, so it doesn’t use XMLHttpRequest(), but Microsoft’s idiosyncratic version thereof, “XMLHTTP”.

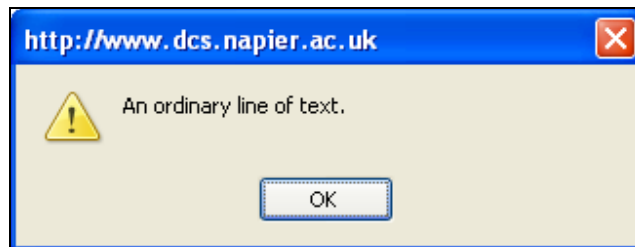
Microsoft doesn’t conform to standards, so JavaScript usually needs to do browser detection and have two sets of code. The box below shows only the JavaScript function.

```
<script type='text/javascript'>
var suck

function popupfile() {
suck=new ActiveXObject("Microsoft.XMLHTTP")
suck.open("GET","text.txt",true)
suck.onreadystatechange=stateChanged
suck.send(null) }

function stateChanged() {
if (suck.readyState==4) {
    alert(suck.responseText); } }
</script>
```

This piece of code does not operate step by step. When the function `popupfile()` is activated from the body of the HTML, it sets up a number of processes as objects. These processes are activated when the function sends “null”. The XMLHTTP line activates the Microsoft ActiveXObject. Then a request is sent to the server to get file “text.txt”. (It could contain anything; it actually contains the words “An ordinary line of text”.) Then it waits, monitoring the status of the XMLHTTP object, until it goes into state 4, indicating that the file has arrived. At that point it pops up the alert box, containing the text of the file.



#### 4. allpop.htm

Now we're going to look at the cross-browser version. This is going to be a little more complicated, but this standard cross-browser solution can be used and reused, and that is some comfort. I share your pain, but this complexity is because Microsoft uses a refusal to standardise as a marketing strategy. We start out by finding out which browser we've arrived in. This first function finds out what the browser is and sorts out an XMLHTTP object for it...

```

function GetXmlHttpRequest() {
var objXMLHttp=null

if (window.XMLHttpRequest) {
    objXMLHttp=new XMLHttpRequest() }
else if (window.ActiveXObject) {
    objXMLHttp=new ActiveXObject("Microsoft.XMLHTTP")
}
return objXMLHttp }

```

So far so good. Whatever the browser, we've activated its XMLHTTP object and given it a standard name to refer to it by. Now the main body of the request, which is similar to the all-IE version: As before, we set up a set of threads, which will "happen" when "send" fires them off. We use the appropriate function for this particular browser (xmlHttp), which we discovered in the function above:

```

var xmlHttp

function popupfile() {
xmlHttp=GetXmlHttpRequest()
if (xmlHttp==null) {
    alert ("Browser does not support HTTP Request")
    return }
xmlHttp.onreadystatechange=stateChanged
xmlHttp.open("GET","text.txt",true)
xmlHttp.send(null)
}

```

A few lines of that routine would be familiar from the example before.

Finally we wait for the text file to come in from the server. When it's in, we pop up the contents in an alert box:

```

function stateChanged() {
    if (xmlHttp.readyState==4) {
        alert(xmlHttp.responseText);
    }
}

```

...so really, the only difference between this general example and the specialised IE example before is the code for finding out which browser we're in and therefore which object we have to use.

You may become better than me, but I find it very difficult to remember all this code. So I make sure I've got a working prototype somewhere and cut and paste from that. Then I hack accordingly.

## 5. within.htm

Eventually, we're going to have to learn how to put imported text into our document. We can't keep on using alert boxes (even though they are fantastic for debugging). This next example shows how this is done

```
<input type='button' onclick='rewriter("here", "banana")'
value='banana' />
```

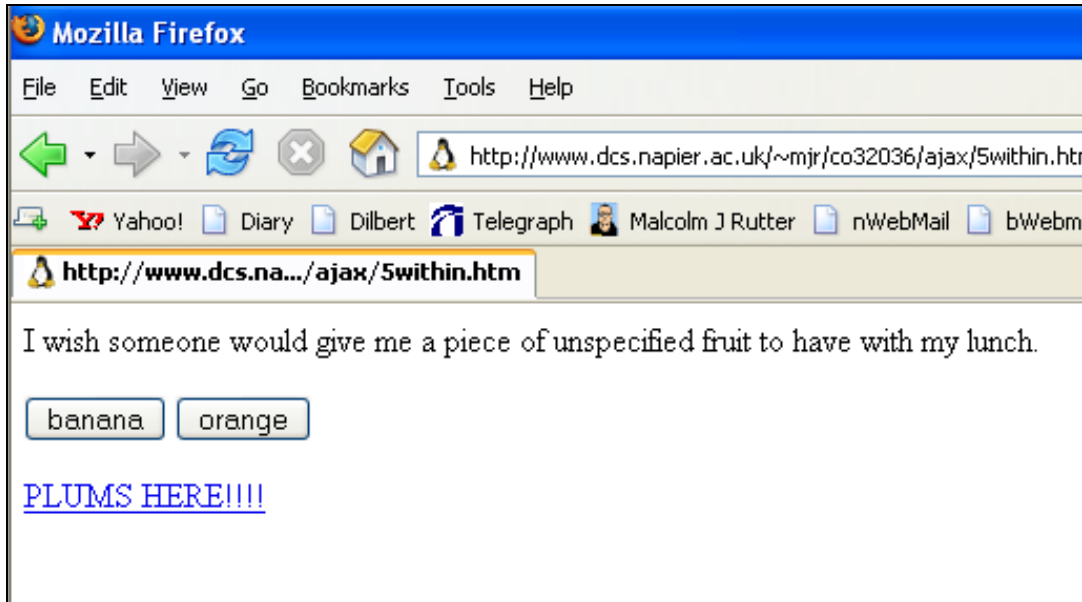
On clicking this button, it calls a function called “rewriter” to put the word “banana” into a place called “here”. Now all we need is a function called “rewriter”. It uses a DOM (Document Object Model) instruction.

```
<script type='text/javascript'>
function rewriter(where, what) {
document.getElementById(where).innerHTML = what;
}
</script>
```

We use the DOM instruction “getElementById” to put text “what” into an HTML element with an ID of “where”. Here is an example of a piece of HTML with an ID for writing into:

```
<p>
I wish someone would give me a piece of
<span id="here"> unspecified fruit </span>
to have with my lunch.
</p>
```

We could use the ID of any element, such as <p> or <div>, but <span> is so useful!



## 6. withinphp.htm

In the next example, we use an external PHP file to supply a value. The PHP programme may be as simple or as complicated as you want (this one is very simple). It may sometimes be more convenient for you to use PHP to access the data system, whatever it is. Popular data systems for the web include MySQL and XML. If you're a Microsoft fan, you may even consider Access, with or without ASP technology.

```
xmlHttp.open("GET", "test.php?action="+what, true)
```

See how we have built up a typical PHP "get"-type parameter-passing request by adding a parameter ("what") to the text of an URL. An example of the entire URL might be: <http://www.eg.com/test.php?action=one>



We aren't teaching PHP here, but if you really are interested in the contents of the simple little PHP file, here it is:

```
<?php  switch($_REQUEST['action']) {
    case 'one':
        echo "grapefruit";
        break;
    case 'two':
        echo "banana";
        break;
    default:
        echo "missed everything!";
        break;
}  ?>
```

In the code, “switch” is PHP for a case statement. The array element `$_REQUEST['action']` accesses that one input parameter in the URL that we chose to give the name of “action”. Meanwhile, “echo” is the programme’s output.

## 7. xml.htm

Lastly, we get and display the root element of an XML file. We have already covered the process of getting a file from a server. The object `responseText` has become `reponseXML` here because we want the browser to treat it as XML.

```
xmlHttp.open("GET", "test.xml", true)
```

```
if (xmlHttp.readyState==4)    {
var xmldoc = xmlHttp.responseXML;
```

We needed a variable to use to refer to our XML document (object), so we’ve used “xmldoc”. We then lock onto the node that I happen to have called ‘napier’ :

```
var root_node =
xmldoc.getElementsByTagName('napier').item(0);
```

Then, with another DOM instruction, we find its first child and feed that data into our web page:

```
document.getElementById('here').innerHTML =  
root_node.firstChild.data;
```

## ***Summary***

What have we learned? We now know a little JavaScript. We can use alert boxes for debugging. We always knew how to read a file from the server; now we can do it without changing web page. We can change some of the text of our page, on the fly. We can pull data out of an XML file and display it.

## ***Tutorial Work***

Find <http://w3schools.com> and do their AJAX course.

Find the directory of example programs on the server. One source is WebCT, another is: <http://www.dcs.napier.ac.uk/~mjr/co32036/ajax/>

Read them and make sense of what you read.

Run them and prove to yourself that they actually go.

Learn through play: Hack them around and make them do something else.

---