At the Forge

JavaScript

Reuven M. Lerner

Abstract

Like the language or hate it, JavaScript and Ajax finally give life to the Web.

About 18 months ago, Web developers started talking about Ajax. No, we weren't suddenly interested in scrubbing our kitchen countertops; rather, we were looking to take advantage of the latest paradigm in Web development. The technology itself wasn't all that new, but the idea that it was a general paradigm Web developers could adopt was new—and when it was given the snappy term Ajax (short for asynchronous JavaScript + XML) by James Paul Garrett, it was only a matter of time before everyone started demanding Ajax Web applications.

The theory behind Ajax is a relatively simple one. JavaScript excels at dynamically changing the HTML pages that Web browsers display. Modern versions of JavaScript are able to make asynchronous HTTP requests to a server. If we combine these two features, we can create desktop-like applications within the browser. Suddenly, the Web is a stable platform for all sorts of new applications. Google Maps was just the beginning; get set for Ajax word processors, spreadsheets, network-administration programs, drawing programs—you name it.

And indeed, we have seen an explosion of Ajax applications in the last year or so. Startups established to create Ajax versions of existing applications already have been bought by companies such as Google. Existing Web sites are scrambling to include Ajax functionality. Book publishers are printing Ajax-related books like they're going out of style. I probably know of at least six toolkits for adding Ajax to applications, and new ones are being released all the time.

Much of the excitement behind Ajax is the freedom it gives designers and developers. Before Ajax, Web applications could be beautiful to look at, but their page-based interfaces were reminiscent of old mainframes, whose applications ran on a page model. What, you want to create an application that is updated incrementally? Sorry, the HTTP/HTML combo means that you either got a new page, and got to enjoy the functionality that it offered, or you stayed on the current one. Every page update had to be accompanied by an HTTP request, and vice versa.

There is no doubt that Ajax applications have a cleaner look and feel to them than old-style Web applications. They feel more natural and responsive, and it's easy to imagine all Web applications looking like this within a few years. This is probably a good thing overall, and I'm looking forward to what the future will bring. In fact, I would guess that within a few years, saying you're an “Ajax developer” will sound as funny as saying you're a “cookie developer”, or a “DOM developer” or even a “database developer”. Just as understanding each of these technologies is now an expected part of being a Web developer, the same is true for Ajax. Yes, this means that Web developers have yet another set of technologies to learn if they want to keep up.
Starting this month, I begin looking at Ajax, with an emphasis on the core of what you'll need to know in order to adjust to this new paradigm. I hope to cover the entire stack, starting with the underlying infrastructure on which Ajax depends—JavaScript, dynamic HTML and XML—moving up to the Ajax techniques themselves, and finally looking at libraries and frameworks that provide prepackaged Ajax functionality.

This month's column begins with the JavaScript language, which sits inside of every major Web browser. By the end of this column, you should have a good idea how to write basic functions in JavaScript and how to ensure that they are called automatically under a variety of different circumstances.

**JavaScript**

As a language, JavaScript has a long and checkered history. It began as a language called LiveScript, which was put inside of Netscape's browsers so as to differentiate them from the competition. The name was changed to JavaScript when Sun Microsystems unveiled its Java language, including applets that sit inside of browsers. The overwhelming excitement over Java applets convinced Netscape to change the name to JavaScript, causing a great deal of confusion. Microsoft then released a largely compatible version of JavaScript in Internet Explorer, called JScript.

After several years in which Web developers dealt with incompatible versions, Netscape had the language standardized by the European organization ECMA. Officially, the language is now known as ECMAScript, but no one really calls it that. The versions in Internet Explorer and Mozilla are now largely compatible with the standard, although there are still differences and issues to work around.

Although JavaScript occasionally has been used in other places, it is overwhelmingly a language that sits inside a Web browser. Its strength is that it allows Web developers to turn static pages of HTML into interactive applications. JavaScript can control every aspect of a Web page, including the contents, forms and design. JavaScript accesses this information via the DOM (document object model, a standard of the World-Wide Web Consortium for describing HTML and XML documents) and the BOM (browser object model, an unstandardized way of getting information about the browser and its objects, such as windows).

I should be up front right now and indicate that I don't particularly like the JavaScript language. Although it has improved substantially in every way since it was first released, a number of aspects really bother me, including the object model and the multiple ways in which undefined values can be represented. That said, disliking JavaScript doesn't change the fact that it sits at the core of Ajax development. Web developers who ignore JavaScript, or who hope that it will simply go away or even be replaced by a different language, are being unrealistic. Rather, we should learn JavaScript, understand its good and bad aspects, and then use it as wisely as possible in our applications.

JavaScript is normally placed inside of the `<script>` tag, inside of an HTML document. In theory, we can put JavaScript inside of any type of HTML document, ignoring the standards. In practice, it is wise to use XHTML, the XML-compliant version of HTML, when working with JavaScript and other Ajax technologies in any serious way. This increases our ability to predict whether a page will work correctly with JavaScript, as well as how it will be rendered by the user's browser. Thus, a simple page containing JavaScript might look like this:
The HTML part of the page is presumably familiar to you. Between the <script> and </script> tags, we have defined a JavaScript function, helloWorld, which pops up a modal dialog box containing our text. The above page works just fine, but the JavaScript itself is never executed. How can we get it to fire?

Perhaps the simplest way is to create a button—an <input> type meant for exactly this task—and have that button then execute our code. For example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <script type="text/javascript">
    function helloWorld() {
        alert("Hello, world!");
    }
    </script>
    <title>Test JavaScript page</title>
</head>
<body>
<p>Hi there!</p>
<p><input type="button" value="Click me!" onclick="helloWorld();" /></p>
</body>
</html>
```

There are several things to notice about the previous code. To begin, we managed to connect the function helloWorld to our button by means of an event handler. This is the way most functions are executed in JavaScript, and it is one of the parts of the language that I most like. You define your functions in the <script> section and then indicate when they should fire using appropriate event handlers.

There is a variety of handlers, all of which begin with the letters “on”, so you can execute a function when someone clicks (onclick), when something is changed (onchange), when an element gets the mouse focus (onfocus) or loses it (onblur), and a number of other possibilities. (Because we're using XHTML, all attributes must be in lowercase. So although it might be tempting to make the event handler more legible by writing onClick, that will invalidate the page.)
We can make this a bit more interesting by personalizing the message:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<script type="text/javascript">
function helloWorld() {
    var username = document.getElementById("username").value;
    if (username == "") {
        alert("Please enter your name in the text field.");
    } else {
        alert("Hello, " + username + "!");
    }
}
</script>
<title>Test JavaScript page</title>
</head>
<body>
<p>Hi there!</p>
<p>Enter your name: <input type="text" id="username" /></p>
<p><input type="button" value="Click me!" onclick="helloWorld();" /></p>
</body>
</html>
```

In this latest version, we added a text field (an `<input>` tag of type text) to the document. The text field has an id attribute, which uniquely identifies it on the page of HTML. Our JavaScript function, meanwhile, has gotten somewhat more complicated. We define a username variable, indicating with the `var` keyword that it is a local variable, rather than a global one. We then ask JavaScript to give us the value of the username element in the current document. Because id attributes are unique, we know there will be at most one element on the page with an ID of username. However, we might misspell or otherwise make a mistake with the username. Or users might not yet have entered their name into the text field, thinking that they should click the button right away. Regardless, we need to test the value of the username variable, so that we won’t foolishly say, “Hello,!” to the user. We thus use an if statement, which works similarly to if in many other C-like languages, putting up a warning note if someone tries to say “hello” without entering a username.

### Modifying the Page

So far, we have seen that JavaScript makes it easy to read values from the page. But part of the magic of JavaScript, and the reason why it sits at the core of Ajax, is that we can modify the contents of the page almost as easily as we can read them. For example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
```
The difference between this version and the previous one is in the else clause of the if statement. After displaying the “hello” alert box, we look for an element whose ID is result. This belongs to a new <p> element on the page, whose contents are currently empty, and which sits just below the button. Note the difference between our two invocations of getElementById. In the case of the username, we want the text that the user has entered, and thus invoke the value method on the returned node. In the case of result, we keep the node itself in the variable, without grabbing any value or other property. That's because we want to modify the result node, adding a text node containing our message.

And indeed, we create the new text node by invoking document.createTextNode, passing it an argument of the text we want to display. Creating the text node does not display it; in order for us to put it on the page, we must attach it to a node that is already being displayed. We will attach it to our result node, using the appendChild method.

Now, each time we click on the click me! button, we not only get an alert, but modify the contents of the page as well.

There is a problem with the above code, however. Because we used the appendChild method to add our text node, we end up creating and appending a new text node once for each click of the button. After clicking the button ten times, for example, we will see our “thanks for clicking” message displayed ten times on the page.
The easiest way to avoid this is to ask the result node if it has any children. If it does, we can assume we have already displayed our result. If no children exist, we safely can go ahead with adding the new text node:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<script type="text/javascript">
function helloWorld() {
  var username = document.getElementById("username").value;
  if (username == "")
    {
      alert("Please enter your name in the text field.");
    }
  else
    {
      alert("Hello, " + username + "!");

      var result = document.getElementById("result")
      if (result.childNodes.length > 0)
        {
          alert("Already has children; not adding to the message");
        }
      else
        {
          var status_text =
          document.createTextNode("Thanks for clicking!");
          result.appendChild(status_text);
        }
  }
}</script>
<title>Test JavaScript page</title>
</head>
<body>
<p>Hi there!</p>
<p>Enter your name: <input type="text" id="username" /></p>
<p><input type="button" value="Click me!" onclick="helloWorld();" /></p>
<p id="result"></p>
</body>
</html>
```

The above, and final, version of our HTML page adds a new if statement after retrieving the result node. Just as we can add new nodes to result, we also can query it to find out if it already has children. When it is first downloaded to the user's browser, the result node has no children. It is empty. So, we can invoke result.childNodes, which returns a list of all children of the result node.

In this case, when we simply want to check to see if any children exist, we invoke the length method on the returned node list. If any number of children exist, we alert the user to the fact that we're not going to add to the message. We will modify the HTML page once, adding our “thanks for clicking” message,
but if the user clicks again, JavaScript will detect the existing text node and not make the changes.

**Conclusion**

JavaScript sits at the heart of the shift toward Ajax applications on the Web, so it is an essential technology for Web developers to understand. JavaScript makes it possible to write functions that read and write the HTML returned by the server. These functions are then invoked by attaching them to predefined events. Next month, we will see how JavaScript allows us to work with stylesheets, which affect the design of a page.