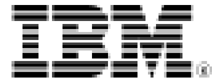


developerWorks®

Mastering Ajax, Part 4: Exploiting DOM for Web response

Convert HTML into an object model to make Web pages responsive and interactive

Skill Level: Introductory

Brett McLaughlin
Author and Editor
O'Reilly Media Inc.

14 Mar 2006

The great divide between programmers (who work with back-end applications) and Web programmers (who spend their time writing HTML, CSS, and JavaScript) is long standing. However, the Document Object Model (DOM) bridges the chasm and makes working with both XML on the back end and HTML on the front end possible and an effective tool. In this article, Brett McLaughlin introduces the Document Object Model, explains its use in Web pages, and starts to explore its usage from JavaScript.

Like many Web programmers, you have probably worked with HTML. HTML is how programmers start to work on a Web page; HTML is often the last thing they do as they finish up an application or site, and tweak that last bit of placement, color, or style. And, just as common as using HTML is the misconception about what exactly happens to that HTML once it goes to a browser to render to the screen. Before I dive into what you might think happens -- and why it is probably wrong -- I want you need to be clear on the process involved in designing and serving Web pages:

1. Someone (usually you!) creates HTML in a text editor or IDE.
2. You then upload the HTML to a Web server, like Apache HTTPD, and make it public on the Internet or an intranet.

© Copyright IBM Corporation 2006

3. A user requests your Web page with a browser like Firefox or Safari.
4. The user's browser makes a request for the HTML to your Web server.
5. The browser renders the page it receives from the server graphically and textually; users look at and activate the Web page.

While this feels very basic, things will get interesting quickly. In fact, the tremendous amount of "stuff" that happens between steps 4 and 5 is what the focus of this article. The term "stuff" really applies too, since most programmers never really consider exactly what happens to their markup when a user's browser is asked to display it:

- Does the browser just read the text in the HTML and display it?
- What about CSS, especially if the CSS is in an external file?
- And what about JavaScript -- again often in an external file?
- How does the browser handle these items and how does it map event handlers, functions, and styles to that textual markup?

It turns out that the answer to all these questions is the Document Object Model. So, without further ado, let's dig into the DOM.

Web programmers and markup

For most programmers, their job ends where the Web browser begins. In other words, once you drop a file of HTML into a directory on your Web server, you usually file it away as "done" and (hopefully) *never really think about it again*! That's a great goal when it comes to writing clean, well-organized pages, too; there's nothing wrong with wanting your markup to display what it should, across browsers, with various versions of CSS and JavaScript.

The problem is that this approach limits a programmer's understanding of what really happens in the browser. More importantly, it limits your ability to update, change, and restructure a Web page dynamically using client-side JavaScript. Get rid of that limitation, and allow even greater interaction and creativity on your Web sites.

What the programmer does

As the typical Web programmer, you probably fire up your text editor and IDE and start to enter HTML, CSS, or even JavaScript. It's easy to think about those tags and selectors and attributes only as little tasks that you do to make a site look just right. But you need to stretch your mind beyond that point -- instead, realize that you are

© Copyright IBM Corporation 2006

organizing your content. Don't worry; I promise this won't turn into a lecture on the beauty of markup, how you must realize the true potential of your Web page, or anything else meta-physical. What you do need to understand is exactly what your role in Web development is.

When it comes to how a page looks, at best you're only able to make suggestions. When you provide a CSS stylesheet, a user can override your style choices. When you provide a font size, a user's browser can alter those sizes for the visually impaired or scale them down on massive monitors (with equally massive resolutions). Even the colors and font faces you choose are subject to the monitor of users and the fonts that users install on their systems. While it's great to do your best in styling a page, that's *not* the largest impact you have on a Web page.

What you do absolutely control is the *structure* of your Web page. Your markup is unchangeable and unalterable and users can't mess with it; their browsers can only retrieve it from your Web server and display it (albeit with a style more in line with the user's tastes than your own). But the organization of that page -- whether this word is inside that paragraph or in the other div -- is solely up to you. When it comes to actually changing your page (which is what most Ajax applications focus on), it's the structure of your page that you operate on. While it's nice to change the color of a piece of text, it's much more dramatic to add text or an entire section to an existing page. No matter how the user styles that section, you work with the organization of the page itself.

What the markup does

Once you realize that your markup is really about organization, you can view it differently. Rather than think that an `h1` causes text to be big, black, and bold, think about an `h1` as a heading. How the user sees that -- and whether they use your CSS, their own, or some combination of the two -- is a secondary consideration. Instead, realize that markup is all about providing this level of organization; a `p` indicates that text is in a paragraph, `img` denotes an image, `div` divides a page up into sections, and so forth.

You should also be clear that style and behavior (event handlers and JavaScript) are applied to this organization *after the fact*. The markup has to be in place to be operated upon or styled. So, just as you might have CSS in an external file to your HTML, the organization of your markup is separate from its style, formatting, and behavior. While you can certainly change the style of an element or piece of text from JavaScript, it's more interesting to actually change the organization that your markup lays out.

As long as you keep in mind that your markup only provides an organization, or framework, for your page, you're ahead of the game. And a little further on, you'll see how the browser takes all this textual organization and turns it into something much more interesting -- a set of objects, each of which can be changed, added to, or

© Copyright IBM Corporation 2006

deleted.

Some additional thoughts on markup

Plain text editing: Right or wrong?
Plain text files are ideal for storing markup, but that doesn't hold true for *editing* that markup. It's perfectly acceptable to use an IDE like Macromedia DreamWeaver -- or the slightly more intrusive Microsoft® FrontPage® -- to work on Web page markup. These environments often offer shortcuts and help in creating Web pages, especially when you're using CSS and JavaScript, each from a file external to an actual page's markup. Many folks still prefer good old Notepad or vi (I confess I'm one of those folk) and that's a great option as well. In either case, the final result is a text file full of markup.

Text over the network: A good thing
As already mentioned, text is a great medium for a document -- like HTML or CSS -- that is transferred over a network hundreds and thousands of times. When I say that the browser has a hard time representing text, that means specifically turning text into the visual and graphical page that users view. This has no bearing on how the browser actually retrieves a page from a Web server; in that case, text is still very much the best option.

The advantages of text markup

Before I discuss the Web browser, it's worth considering why plain text is *absolutely* the best choice for storing your HTML (for more on this, see Some additional thoughts on markup). Without getting into the pros and cons, simply recall that your HTML is sent across a network to a Web browser every time the page is viewed (putting caching and so forth aside for simplicity's sake). There's simply no more efficient approach to take than passing along text. Binary objects, graphical representations of the page, a re-organized chunk of markup ... all of these are harder to send across the network than plain text files.

Add to that the value that a browser adds to the equation. Today's browsers allow users to change the size of text, scale images, download the CSS or JavaScript for a page (in most cases), and a lot more -- this all precludes sending any kind of graphical representation of the page to the browser. Instead, the browser needs the raw HTML so it can apply whatever processing to the page in the browser rather than trusting the server to handle that task. In the same vein, separating CSS from JavaScript and separating those from the HTML markup requires a format that is easy to, well, separate. Text files again are a great way to do just this.

Last, but not least, remember that the promise of new standards like HTML 4.01 and XHTML 1.0 and 1.1 separates content (the data in your page) from presentation and style (usually applied by CSS). For programmers to separate their HTML from their CSS, then force a browser to retrieve some representation of a page that melds these all back together, defeats much of the advantage of these standards. To keep

© Copyright IBM Corporation 2006

these disparate parts separate all the way down to the browser allows the browser the most flexibility in getting the HTML from the server.

A closer look at Web browsers

For some of you, everything you've read so far may be droll review of your role in the Web development process. But when it comes to what the Web browser does, many of the most savvy Web designers and developers often don't realize what actually goes on "under the hood." I'll focus on that in this section. And don't worry, code is coming soon, but hold off on your coding impatience because really understanding exactly what a Web browser does is essential to your code working correctly.

The disadvantages of text markup

Just as text markup has terrific advantages for a designer or page creator, it also has rather significant disadvantages for a browser. Specifically, browsers have a very difficult time directly representing text markup to a user visually (for more on this, see Some additional thoughts on markup). Consider these frequent browser tasks:

- Apply CSS styles -- often from multiple stylesheets in external files -- to markup based on the type of element, its class, its ID, and its position in the HTML document.
- Apply styles and formatting based on JavaScript code -- also often in external files -- to different parts of the HTML document.
- Change the value of form fields based on JavaScript code.
- Support visual effects like image rollovers and image swapping based on JavaScript code.

The complexity isn't in coding these tasks; it's fairly easy to do each of these things. The complexity comes from the browser actually carrying out the requested action. If markup is stored as text and, for example, you want to center the text (`text-align: center`) in a `p` element in the `center-text` class, how do you accomplish that?

- Do you add inline styling to the text?
- Do you apply the styling to the HTML text in the browser and just keep up with which content to centered or not center?
- Do you apply unstyled HTML and then apply format after the fact?

These very difficult questions are why few people code browsers these days. (Those who do should receive a hearty "Thanks!")

© Copyright IBM Corporation 2006

Clearly, plain text isn't a great way to store HTML for the browser even though text was a good solution for getting a page's markup in the first place. Add to this the ability for JavaScript to *change* the structure of a page and things really get tricky. Should the browser rewrite the modified structure to disk? How can it keep up with what the current stage of the document is?

Clearly, text isn't the answer. It's difficult to modify, clumsy to apply styles and behavior to, and ultimately bears little likeness to the dynamic nature of today's Web pages.

Moving to a tree view

The answer to this problem -- at least, the answer chosen by today's Web browsers -- is to use a tree structure to represent HTML. Take a look at Listing 1, a rather simple and boring HTML page represented as text markup.

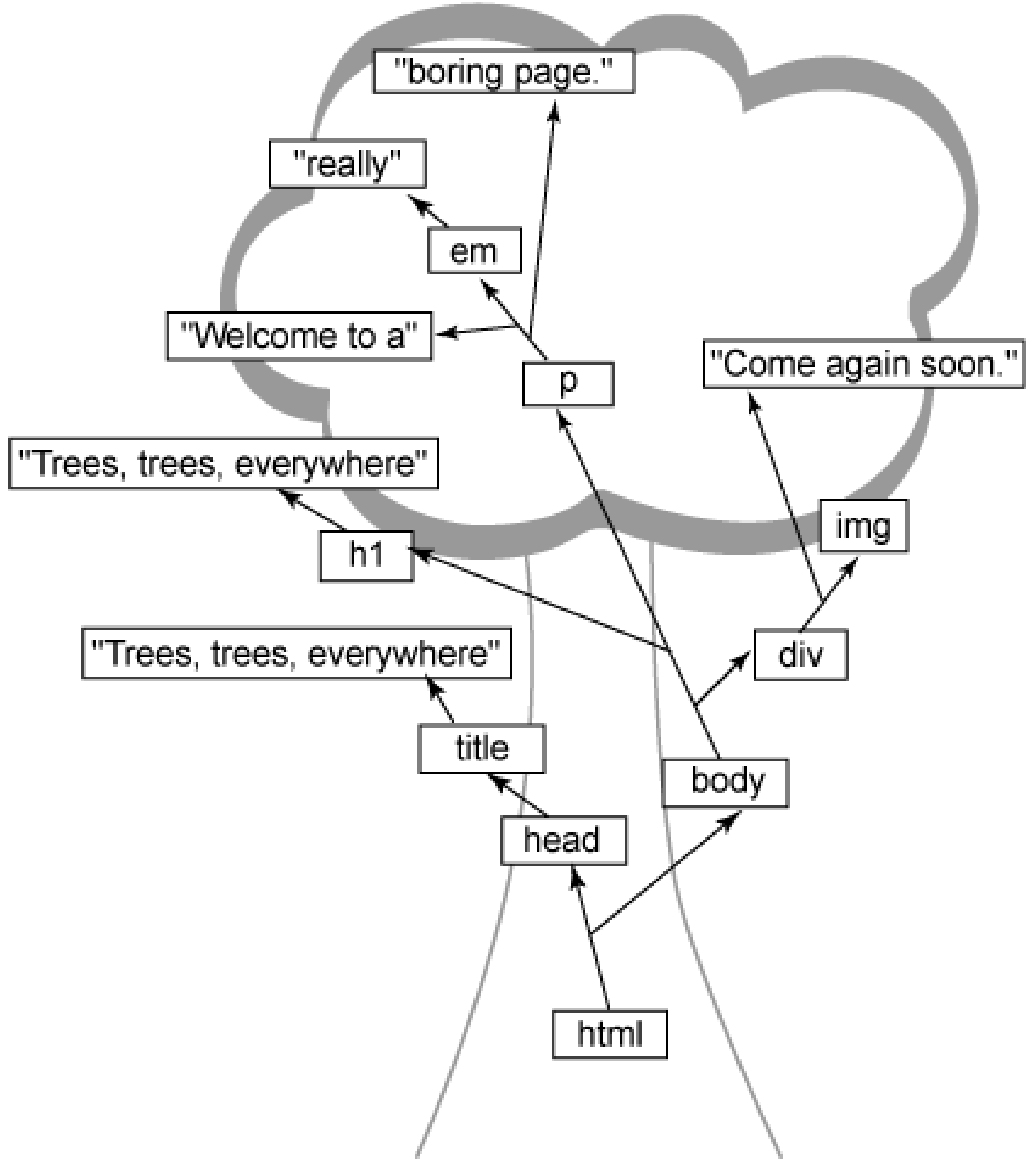
Listing 1. Simple HTML page in text markup

```
<html>
 <head>
  <title>Trees, trees, everywhere</title>
 </head>
 <body>
  <h1>Trees, trees, everywhere</h1>
  <p>Welcome to a <em>really</em> boring page.</p>
  <div>
    Come again soon.
    <img src="come-again.gif" />
  </div>
 </body>
</html>
```

The browser takes this and converts it into a tree-like structure, as in Figure 1.

Figure 1. The tree view of Listing 1

© Copyright IBM Corporation 2006



I made a few very minor simplifications to keep this article on track. Experts in the DOM or XML will realize that whitespace can have an effect on how text in a document is represented and broken up in the Web browser's tree structure. Getting into this does little but confuse the matter, so if you know about the effect of whitespace, great; if not, read on and don't worry about it. When it becomes an issue, you'll find out all you need at that time.

Other than the actual tree background, the first thing you might notice here is that everything in the tree begins with the outer-most, containing element of HTML, which is the `html` element. This is called the *root element* in keeping with the tree

© Copyright IBM Corporation 2006

metaphor. So even though this is at the bottom of the tree, ialways begin here when you look at and analyze trees. If it helps, you're welcome to flip the whole thing upside down, although that does stretch the tree metaphor a bit.

From the root flow out lines that show the relationships between different pieces of markup. The `head` and `body` elements are *children* of the `html` root element; `title` is a child of `head` and then the text "Trees, trees, everywhere" is a child of `title`. The entire tree is organized like this until the browser gets a structure similar to what you see in Figure 1.

A few additional terms

To carry on with the tree metaphor, `head` and `body` are said to be *branches* of `html`. They are branches because they in turn have children of their own. When you reach the extremities of the tree, you'll run into mostly text such as "Trees, trees, everywhere" and "really." These are often referred to *leaves* because they have no children of their own. You don't need to memorize all these terms and it is often easier to just visualize the tree structure when you try to figure out what a particular term means.

The value of objects

Now that you have some basic terminology in hand, it's time to focus more on those little rectangles with element names and text inside (Figure 1). Each rectangles is an object; that's where the browser resolves some of these problems with text. By using objects to represent each piece of the HTML document, it becomes very easy to change the organization, apply styles, allow JavaScript to access the document, and much more.

Object types and properties

Each possible type of markup gets its own object type. For instance, elements in your HTML are represented by an `Element` object type. The text in your document is represented by a `Text` type, attributes are represented by `Attribute` types, and right on down the line.

So the Web browser not only gets to use an object model to represent your document -- getting away from having to deal with static text -- but can immediately tell what something is by its object type. The HTML document is parsed and turned into the set of objects like you saw in Figure 1 and then things like angle brackets and escape sequences (for example, using < for < and > for >) no longer become an issue. This makes the job of the browser, at least after it parses the input HTML, much easier. The operation to figure out whether something is an element or an attribute and then determine what to do with that type of object is simple.

© Copyright IBM Corporation 2006

By using objects, the Web browser can then change those objects' properties. For example, each element object has a parent and a list of children. So adding a new child element or text is simply a matter of adding a new child to an element's list of children. These objects also have a `style` property, so it becomes trivial to change the style of an element or piece of text on the fly. For example, you might change the height of a `div` using JavaScript like this:

```
someDiv.style.height = "300px";
```

In other words, the Web browser can very easily change the appearance and structure of the tree using object properties like this. Compare this to the complicated sorts of things that the browser must do if it represented the page as text internally; every change of property or structure requires the browser to rewrite the static file, re-parse it, and re-display it on the screen. All of this becomes possible with objects.

At this point, take time to pull open some of your HTML documents and sketch them out as trees. While that seems an unusual request -- especially from an article that contains very little code -- you will need to become familiar with the structure of these trees if you want to be able to manipulate them.

In the process, you'll probably find some oddities. For example, consider the following situations:

- What happens to attributes?
- What about text that is broken up with elements, like `em` and `b`?
- And what about HTML that isn't structured correctly (like when a closing `p` tag is missing)?

Once familiar with these sorts of issues, you'll understand the next few sections a lot better.

Strict is sometimes good

If you tried the exercise I just mentioned, you probably found some of the potential troubles for a tree-view of your markup (if you didn't exercise, just take my word for it!). In fact, you'll find several of these in Listing 1 and Figure 1, starting with the way the `p` element was broken up. If you ask the typical Web developer what the text content of the `p` element is, the most common answer would be, "Welcome to a really boring Web page." If you compare this with Figure 1, you'll see that this answer -- although logical -- isn't at all correct.

It turns out that the `p` element has *three* different child objects and none contain all of the text "Welcome to a really boring Web page." You'll find parts of that text, like

© Copyright IBM Corporation 2006

"Welcome to a " and " boring Web page", but not all of it. To understand this, remember that everything in your markup has to be turned into an object of some type.

Furthermore, the order does matter! Can you imagine how users would respond to a Web browser if it showed the correct markup, but in a different ordering than you provided in your HTML? Paragraphs were sandwiched in between titles and headings, even when that's not how you organized the document yourself? Obviously, the browser must preserve the order of elements and text.

In this case, the `p` element has three distinct parts:

- The text that comes before the `em` element
- Tthe `em` element itself
- The text that comes after the `em` element

If you mix this ordering up, you might apply the emphasis to the wrong portion of text. To keep this all straight, the `p` element has three object children in the order that those things appeared in the HTML in Listing 1. Further, the emphasized text "really" *isn't* a child element of `p`; it is a child of `em` which is a child of `p`.

It is very important for you to understand this concept. Even though the "really" text will probably display along with the rest of the `p` element's text, it still is a direct child of the `em` element. It can have different formatting from the rest of the `p` and can be moved around independently of the rest of the text.

To help cement this in your mind, try to diagram the HTML in Listings 2 and 3, making sure you keep text with its correct parent (despite how that text might end up looking on screen).

Listing 2. Markup with slightly tricky element nesting

```
<html>
 <head>
  <title>This is a little tricky</title>
 </head>
 <body>
  <h1>Pay <u>close</u> attention, OK?</h1>
  <div>
   <p>This p really isn't <em>necessary</em>, but it makes the
      <span id="bold-text">structure <i>and</i> the organization</span>
      of the page easier to keep up with.</p>
  </div>
 </body>
</html>
```

Listing 3. Even trickier nesting of elements

```
<html>
 <head>
  <title>Trickier nesting, still</title>
 </head>
 <body>
  <div id="main-body">
   <div id="contents">
    <table>
     <tr><th>Steps</th><th>Process</th></tr>
     <tr><td>1</td><td>Figure out the <em>root element</em>.</td></tr>
     <tr><td>2</td><td>Deal with the <span id="code">head</span> first,
         as it's usually easy.</td></tr>
     <tr><td>3</td><td>Work through the <span id="code">body</span>.
         Just <em>take your time</em>.</td></tr>
    </table>
   </div>
   <div id="closing">
    This link is <em>not</em> active, but if it were, the answers
    to this <a href="answers.html"><img src="exercise.gif" /></a> would
    be there. But <em>do the exercise anyway!</em>
   </div>
  </div>
 </body>
</html>
```

You'll find the answers to these exercises in the GIF files tricky-solution.gif in Figure 2 and trickier-solution.gif in Figure 3 at the end of this article. Don't peek until you take the time to work these out for yourself. It will help you understand how strictly rules apply to organizing the tree and really help you in your quest to master HTML and its tree structure.

What about attributes?

Did you run across any problems as you tried to figure out what to do with attributes? As I mentioned, attributes do have their own object type, but an attribute is really not the child of the element it appears on -- nested elements and text are not at the same "level" of an attribute and you'll notice that the answers to the exercises in Listings 2 and 3 do not have attributes shown.

Attributes are in fact stored in the object model that a browser uses, but they are a bit of a special case. Each element has a list of attributes available to it, *separate* from the list of child objects. So a `div` element might have a list that contained an attribute named "id" and another named "class."

Keep in mind that attributes for an element must have unique names; in other words, an element cannot have two "id" or two "class" attributes. This makes the list very easy to keep up with and to access. As you'll see in the next article, you can simply call a method like `getAttribute("id")` to get the value of an attribute by its name. You can also add attributes and set (or reset) the value of existing attributes with similar method calls.

It's also worth pointing out that the uniqueness of attribute names makes this list different than the list of child objects. A `p` element could have multiple `em` elements

© Copyright IBM Corporation 2006

within it, so the list of child objects can contain duplicate items. While the list of children and the list of attributes operate similarly, one can contain duplicates (the children of an object) and one cannot (the attributes of an element object). Finally, only elements can have attributes, so text objects have no lists attached to them for storing attributes.

Sloppy HTML

Before I move on, one more topic is worth some time when it comes to how the browser converts markup to a tree representation -- how a browser deals with markup that is not well-formed. *Well-formed* is actually a term largely used in XML and means two basic things:

- Every opening tag has a matching closing tag. So every `<p>` is matched in the document by a `</p>`, every `<div>` by a `</div>`, and so forth.
- The innermost opening tag is matched with the innermost closing tag, then the next innermost opening tag by the next innermost closing tag, and so forth. So `<b><i>bold and italics</b></i>` would be illegal because the innermost opening tag -- `<i>` -- is improperly matched with the innermost closing tag -- `<b>`. To make this well-formed, you would need to switch *either* the opening tag order *or* the closing tag order. (If you switched both, you'd still have a problem).

Study these two rules closely. They are both rules that not only increase the simple organization of a document, but they also remove ambiguity. Should bolding be applied first and then italics? Or the other way around? If it seems that this ordering and ambiguity is not a big deal, remember that CSS allows rules to override other rules so if, for example, the font for text within `b` elements was different than the font for within `i` elements, the order in which formatting was applied becomes very important. Therefore, the well-formedness of an HTML page comes into play.

In cases where a browser receives a document that is not well-formed, it simply does the best it can. The resulting tree structure is at best an approximation of what the original page author intended and at worst something completely different. If you ever loaded your page in a browser and saw something completely unexpected, you might have viewed the result of a browser trying to guess what your structure should be and doing the job poorly. Of course, the fix to this is pretty simple: Make sure your documents are well-formed! If you're unclear on how to write standardized HTML like this, consult the Resources for help.

Introducing the DOM

So far you've heard that browsers turn a Web page into an object representation and maybe you've even guessed that the object representation is a DOM tree. DOM

© Copyright IBM Corporation 2006

stands for Document Object Model and is a specification available from the World Wide Web Consortium (W3C) (you can check out several DOM-related links in the Resources).

More importantly though, the DOM defines the objects' types and properties that allow a browser to represent markup. (The next article in this series focuses on the specifics of using the DOM from your JavaScript and Ajax code.)

The document object

First and foremost, you will need to access the object model itself. This is remarkably easy; to use the built-in `document` variable in any piece of JavaScript code running on your Web page, you can write code like this:

```
var domTree = document;
```

Of course, this code is pretty useless in and of itself, but it does demonstrates that every Web browser makes the `document` object available to JavaScript code and that the object represents the complete tree of markup (Figure 1).

Everything is a node

Clearly, the `document` object is important, but is just the beginning. Before you can go further, you need to learn another term: *node*. You already know that each bit of markup is represented by an object, but it's more than just any object -- it's a specific type of object, a DOM node. The more specific types -- like text, elements, and attributes -- extend from this basic node type. So you have text nodes, element nodes, and attribute nodes.

If you've programmed much in JavaScript, it should occur to you that you might already be using DOM code. If you followed this Ajax series thus far, then you've *definitely* used DOM code for some time now. For example, the line `var number = document.getElementById("phone").value;` uses the DOM to find a specific element and then to retrieve the value of that element (in this case, a form field). So even if you didn't realize it, you used the DOM every time you typed `document` into your JavaScript code.

To refine the terms you've learned then, a DOM tree is a tree of objects, but more specifically it is a tree of *node* objects. In Ajax applications -- or any other JavaScript -- you can work with those nodes to create such effects as removing an element and its content, highlighting a certain piece of text, or adding a new image element. Since this all occurs on the client side (code that runs in your Web browser), these effects take place immediately without communication with the server. The end result is often an application that feels more responsive because things on the Web page change without long pauses while a request goes to a server and a response

© Copyright IBM Corporation 2006

is interpreted.

In most programming languages, you need to learn the actual object names for each type of node, learn the properties available, and figure out about types and casting; but none of this is necessary in JavaScript. You can simply create a variable and assign it the object you want (as you've already seen):

```
var domTree = document;
var phoneNumberElement = document.getElementById("phone");
var phoneNumber = phoneNumberElement.value;
```

There are no types and JavaScript handles creating the variables and assigning them the correct types as needed. As a result, it becomes fairly trivial to use the DOM from JavaScript (a later article focuses on the DOM in relation to XML and things are a little trickier).

In conclusion

At this point, I'm going to leave you with a bit of a cliffhanger. Obviously, this hasn't been a completely exhaustive coverage of the DOM; in fact, this article is little more than an introduction to the DOM. There is more to DOM than I've shown you today!

The next article in this series will expand upon these ideas and dive more into how you can use the DOM in your JavaScript to update Web pages, make changes to your HTML on the fly, and create a more interactive experience for your user. I'll come back to DOM once again in later articles which focus on using XML in your Ajax requests. So become familiar with the DOM; it's a major part of Ajax applications.

It would be pretty simple to launch into more of the DOM at this point, detailing how to move within a DOM tree, get the values of elements and text, iterate through node lists, and more, but that would probably leave you with the impression that the DOM is about code -- it's not.

Before the next article, try to think about tree structures and work through some of your own HTML to see how a Web browser would turn that HTML into a tree view of the markup. Also, think about the organization of a DOM tree and work through the special cases discussed in this article: attributes, text that has elements mixed in with it, elements that *don't* have text content (like the `img` element).

If you get a firm grasp of these concepts and then learn the syntax of JavaScript and the DOM (in the next article), it will make crafting responsiveness much easier.

And don't forget, here are the answers to Listings 2 and 3 -- they are also included with the sample code!

© Copyright IBM Corporation 2006

Figure 2. The answer to Listing 2

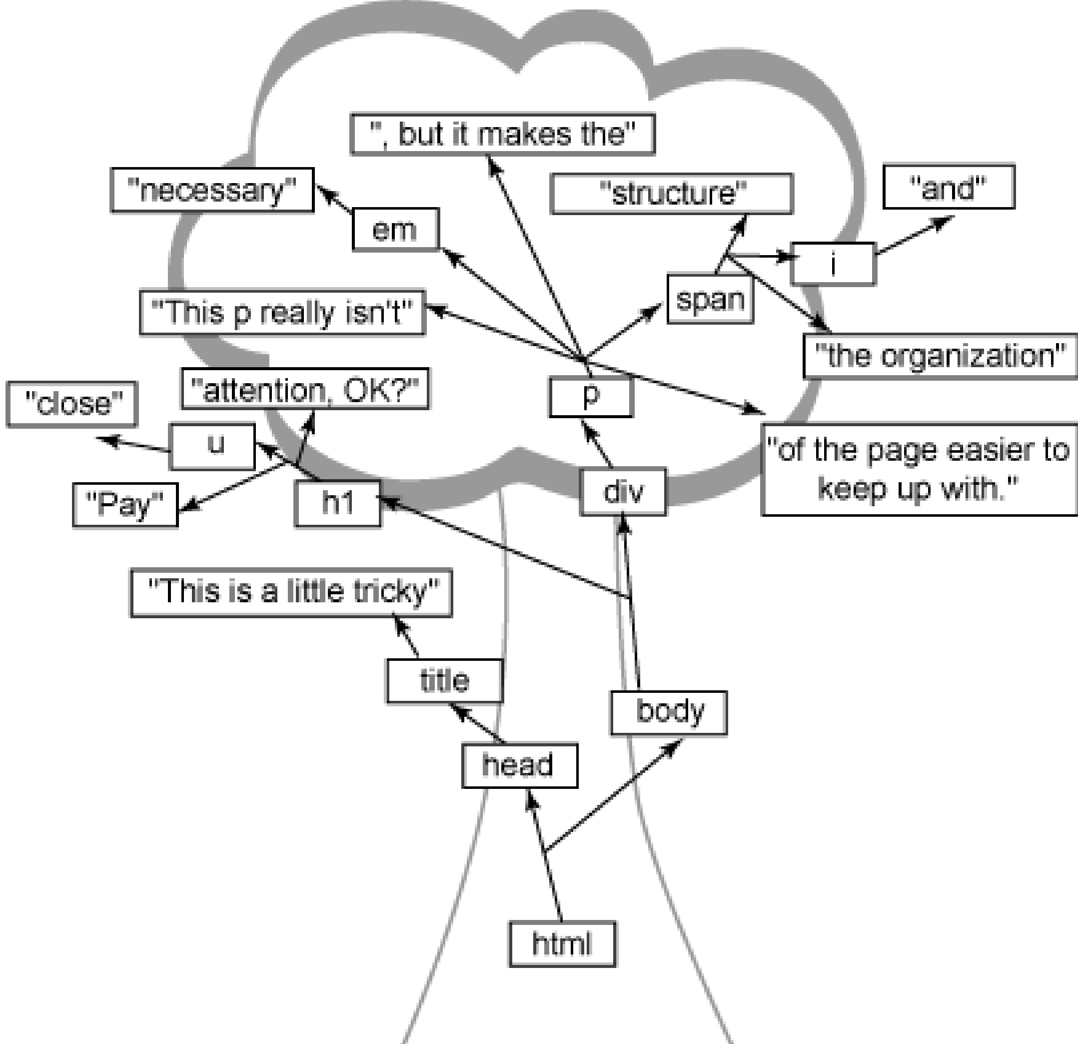
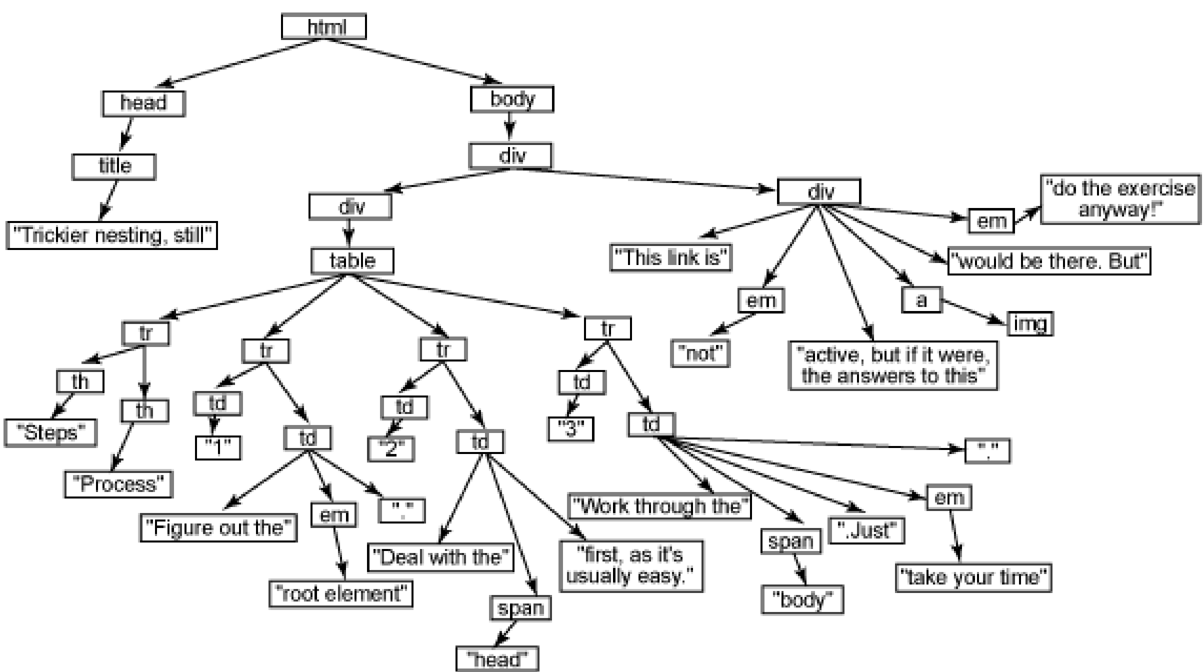


Figure 3. The answer to Listing 3

© Copyright IBM Corporation 2006



© Copyright IBM Corporation 2006

Resources

Learn

- Explore the earlier articles in this developerWorks series introducing Ajax:
    - "Introduction to Ajax: Understanding Ajax, a productive approach to building Web sites, and how it works:" Part 1 demonstrates how Ajax component technologies work together and unveils the central concepts of Ajax, including the `XMLHttpRequest` object (December 2005).
    - "Make asynchronous requests with JavaScript and Ajax: Use XMLHttpRequest for Web requests:" Part 2 shows you how to create `XMLHttpRequest` instances in a cross-browser way, construct and send requests, and respond to the server (January 2006).
    - "Advanced requests and responses in Ajax:" Part 3 demonstrates how standard Web forms perform with Ajax and shows you how to master your understanding of HTTP status codes, ready states, and the `XMLHttpRequest` object (February 2006).
- "Use Ajax with WebSphere Portal" (developerWorks, June 2006) to improve portal performance, create a cleaner portal application architecture, and -- most important -- give your users a much more responsive portal.
- The DOM Home Page at the World Wide Web Consortium: Visit the starting place for all things DOM-related.
- The DOM Level 3 Core Specification: Define the core Document Object Model, from the available types and properties to the usage of the DOM from various languages.
- The ECMAScript language bindings for DOM: If you're a JavaScript programmer and want to use the DOM from your code, this appendix to the Level 3 Document Object Model Core definitions will interest you.
- "Ajax for Java developers: Build dynamic Java applications:" This look at Ajax from the server side using a Java perspective, introduces a groundbreaking approach to creating dynamic Web application experiences (developerWorks, September 2005).
- "Ajax for Java developers: Java object serialization for Ajax:" Walk through five approaches to Java object serialization and examine how to send objects over the network and interact with Ajax (developerWorks, October 2005).
- "Call SOAP Web services with Ajax, Part 1: Build the Web services client:" Dig into this fairly advanced article on integrating Ajax with existing SOAP-based Web services; it shows you how to implement a Web browser-based SOAP Web services client using the Ajax design pattern (developerWorks, October

© Copyright IBM Corporation 2006

2005).

- "Ajax: A New Approach to Web Applications:" Read the article that coined the Ajax moniker -- it's required reading for all Ajax developers.
- HTTP status codes: Review this complete list of status codes from the W3C.
- *Head Rush Ajax* by Elisabeth Freeman, Eric Freeman, and Brett McLaughlin (February 2006, O'Reilly Media, Inc.): Load the ideas in this article into your brain, Head First style.
- *Java and XML*, Second Edition by Brett McLaughlin (August 2001, O'Reilly Media, Inc.): Check out the author's discussion of XHTML and XML transformations.
- *JavaScript: The Definitive Guide* by David Flanagan (November 2001, O'Reilly Media, Inc.): Dig into extensive instruction on working with JavaScript and dynamic Web pages. The upcoming edition adds two chapters on Ajax.
- *Head First HTML with CSS & XHTML* by Elizabeth and Eric Freeman (December 2005, O'Reilly Media, Inc.): Peruse this complete source for learning XHTML, CSS, and how to pair the two.
- developerWorks Web architecture zone: Expand your Web-building skills.
- developerWorks technical events and Webcasts: Stay current with these software briefings for technical developers.

Get products and technologies

- IBM trial software: Build your next development project with software available for download directly from developerWorks.

Discuss

- developerWorks blogs: Get involved in the developerWorks community.

About the author

Brett McLaughlin

Brett McLaughlin has worked in computers since the Logo days.(Remember the little triangle?) In recent years, he's become one of the most well-known authors and programmers in the Java and XML communities. He's worked for Nextel Communications, implementing complex enterprise systems; at Lutris Technologies, actually writing application servers; and most recently at O'Reilly Media, Inc., where he continues to write and edit books that matter. Brett's upcoming book, *Head Rush Ajax*, brings the award-winning and innovative Head First approach to Ajax, along with bestselling co-authors, Eric and Beth

© Copyright IBM Corporation 2006

Freeman. His last book, *Java 1.5 Tiger: A Developer's Notebook*, was the first book available on the newest version of Java technology and his classic *Java and XML* remains one of the definitive works on using XML technologies in the Java language.

© Copyright IBM Corporation 2006