<chapter> 2

An Orientation to JavaScript

CONTENTS>>

Writing JavaScript

As you saw in Chapter 1, "Jump-Starting JavaScript," JavaScript goes into an HTML page. However, you do not write JavaScript with the same abandon as you do HTML. Very specific and apparently minor differences exist between how HTML can be written and how JavaScript can be written. While the differences might appear to be minor or even trivial, if the rules for writing JavaScript are not followed, you can run into glitches. This chapter examines the nuances of JavaScript so that when you start writing your own scripts, you'll have all of the basics clear in your mind.

HTML is a markup language, and JavaScript is a programming or scripting language. HTML describes what is to be presented on a page, and JavaScript dynamically changes what is on an HTML page (among other tasks.) Both use code. HTML's

code is in a series of angle brackets that describe how to treat the material between the opening and closing brackets. JavaScript is a set of statements and functions that does something in an HTML page. JavaScript can refer to and alter objects described by HTML.

Case Sensitivity

You can write HTML tags in just about any way you want, as long as you spell the tags correctly and remember to include the arrow bracket around the tags. For example, the following little page will work just fine in HTML:

```
<hTmL>
<heaD>
<Title>Do it your way</tITLE>
</HEAD>
<BoDY Bgcolor="HotPink">
<ceNTer>
<h1>
HTML is CaSe InSeNsItIvE!
</H1>
</bODY>
</HTML>
```

Just about every non–case-sensitive combination of characters that you can imagine has been put into that page. The opening tags of a container are in one case combination, and the closing tags are in another. Tags in one case are duplicated with tags in another case. For HTML, that's no problem. You don't have to pay attention to case at all.

JavaScript is the opposite. You have to pay attention to the cases of everything that you type in JavaScript because it is case-sensitive. The HTML around the script need not be case-sensitive, but the JavaScript itself must be. Consider the following examples. The first follows the rules of case sensitivity, and the second one does not.

```
<html>
<head>
<title>Case Sensitive</title>
<script language="JavaScript">
alert("Pay attention to your cases!");
</script>
</head>
<body bgcolor="moccasin">
<p>
<h1>Just in case!</h1>
</p>
</body>
</html>
```

When you load the page, you will see an alert message telling you to pay attention to your cases. As soon as you click the OK button on the alert box, the rest of the page appears with the message "Just in case." Now, look at this next script to see if you can tell where the error lies. It is slightly different from the first—only the *a* in "alert" has been changed so that it is "Alert." Just that little change will invalidate the JavaScript code. Launch the page with the capital *A*, and see what happens.

```
<html>
<head>
<title>Case Sensitive</title>
<script language="JavaScript">
Alert("Pay attention to your cases!");
</script>
</head>
<body bgcolor="moccasin">
<p>
<h1>Just in case!</h1>
</p>
</body>
</html>
```

As you saw, the page didn't crash and burn. It just ignored the JavaScript and went on and put up the page on the screen. I would rather see an error message issued so that I could see any problems that arise, but neither of the newest versions of the browsers indicated any trouble at all. (In Netscape Navigator 4.7, a little error message blinks in the lower-left corner, but it happens so fast that you cannot tell that your script has an error.) Debugging JavaScript is often a matter of ***not seeing*** what you expect on the screen rather than seeing any clue that you've coded your script incorrectly. However, ignoring case sensitivity is likely to be one bug in the code that you should suspect immediately.

For the ***most part***, JavaScript is typed in lowercase fonts, but you will find many exceptions to that rule. In Chapter 1, you might have noticed the use of `Math.floor` along with `toString` in one of the scripts. Both of those words use a combination of upper- and lowercase fonts: intercase. `Object`, `Math`, `Date`, `Number`, and `RegExp` are among the objects that use case combinations as well. Properties such as `innerHeight`, `outerWidth`, and `isFinite`, likewise, are among the many other JavaScript terms using case combinations.

You also might run into cases differences in HTML and JavaScript. Event-related attributes in HTML such as `onMouseOver`, `onMouseOut`, and `onClick` are spelled with a combination of upper- and lowercase characters by convention, but, in JavaScript, you must use all lowercase on those same terms. Hence, you will see `.onmouseover`, `.onmouseout`, and `.onclick`.

Another area of case sensitivity in JavaScript can be found in naming variables and functions. You can use any combination of upper- and lowercase characters that you want in a function or variable name, as long as it begins with an ASCII letter, dollar sign, or underscore. Functions are names that you give to a set of other statements or commands. (See an introduction to functions in Chapter 1.) Variables are names that you give to containers that hold different values. For example, the variable `customers` might contain the words "John Davis" or "Sally Smith." Variables can contain words or numbers. (See the next chapter for a more detailed discussion of variables.) When the function or variable is given a name, you must use the same set of upper- and lowercase characters that you did when you declared the variable or functions. For example, the following script uses a combination of characters in both variables and function. When the function is fired, the name must be spelled as it is in the definition.

VarFuncCase.html

```
<html>
<head>
<title>Cases in Variables and Functions</title>
<script language="JavaScript">
var Tax=.05
function addTax(item) {
var Total=item + (item * Tax);
var NewTotal=Math.floor(Total);
var Fraction=Math.round(Total *100)%100;
if (Fraction<10) {
Fraction = "0" + Fraction;
}
Total=NewTotal +"." + Fraction;
alert("Your total is $" + Total);
}
</script>
</head>
<body bgcolor="palegreen">
<center><h2>
<a href=# onClick="addTax(7.22)";> Click for total </a>
</body>
</html>
```

Several variables and two functions (the `alert` function is built in and so has a name already—alert) are included in the script, but notice that all of the variable names and function references use the same combination of upper- and lowercase characters. The string message for the `alert` function reads, `"Your total is $" + Total);`. The first use of `total` is part of a message (string literal) and is not a variable in this case. The `Total` attached to the end of the alert message, however, is

a variable, and it uses the uppercase first letter as the variable does when it is defined. Likewise, the argument in the function (`item`) is always referenced in lowercase because it is initially written in lowercase. The variable declaration lines beginning with `var` signal the initial creation of a variable, and the case configuration used in those lines is the configuration that must be used throughout the page in reference to a given variable. Chapter 3, "Dealing with Data and Variables," explains developing variables in detail.

Figure 2.1 shows what you will see when the page loads and you click the link text. If you want a link to launch a JavaScript function, you can use a "dummy" link by inserting a pound sign (`#`) where the URL usually is placed. Then, by adding an event handler, you can launch the function. Try changing the value in the `addTax()` function to see what you get. Also, see what happens when you change `addTax` to `ADDTAX()`.

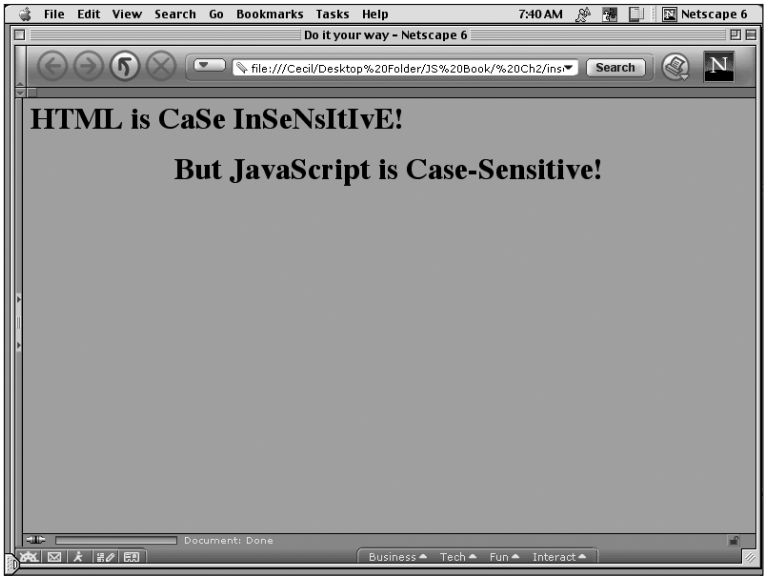

FIGURE 2.1

None of HTML is case-sensitive, but virtually all of JavaScript is.

Entering Comments

Comments are messages to yourself and others who are working to develop a JavaScript program with you. They serve to let you know what the following lines of code do or, if incomplete, what you want them to do. Comments in JavaScript are entered by prefacing a line with double forward slashes (`//`). When the code is parsed in the browser, all of the lines beginning with the double slashes are

ignored. For example, the following code segment shows a reminder to add tax to an item in an e-business application:

```
//Include a variable to add taxes
tax= .06
//Add the tax to the taxable item
item += item * tax
```

The bigger and more complex your scripts become, the more you will need to have well-commented code. Comments in code become even more crucial when you are working with a team to create a web site and others need to know what your code is doing. In this book, you will see comments throughout the code in the larger scripts to point out different elements. In shorter scripts, the comments are in the text of the book, a luxury that you will not have in your own coding. Remember to comment your code, and you will see that you can save a good deal of time reinventing a solution that is already completed.

The Optional Semicolon

Several languages that look a lot like JavaScript require a semicolon after lines. For example, Flash ActionScript and PHP (see Chapter 14, "Using PHP with JavaScript," and Chapter 18, "Flash ActionScript and JavaScript") both require semicolons. Likewise, compiled languages such as C++ and Java require semicolons at the end of lines. JavaScript made semicolons optional.

So, the question is, do you really need the semicolon? JavaScript places "invisible" semicolons at the end of each line, and by placing a ***visible semicolon***, you can better see what's going on. For debugging your program, the semicolons alert you to where a line ends; if you did not intend a line to end where you put a semicolon, you can better see the error. Hence, the answer to the question of whether you should include semicolons is "yes."

Semicolons go at the end of lines that do not end in a curly brace or to separate variable declarations on the same line. For example, the following two code segments show where semicolons may optionally be placed.

```
function findIt() {
      if(x="searchWord") {
            document.formA.elementA.value=x;
      }
}
```

Because four of the five lines end in a curly brace, only the third line optionally can have a semicolon. On the other hand, in a list of variable definitions, you can place a semicolon at the end of every line.

```
var alpha="Apples";
var beta= alpha + "Oranges";
var gamma= Math.sqrt(omega);
var delta= 200/gamma;
```

Older Browsers

At the time of this writing, Netscape Navigator 6.01 is in general release for both Windows and Macintosh operating systems, and Internet Explorer has a Version 6 in public preview for Windows and is in Version 5.5 on the Macintosh. By the time this book is published, both major browsers will most likely have Version 6 as their standard browser. Keeping in mind that the browsers are the interpreters for JavaScript, the version of browser that others use to view your scripts is very important. Version 3 browsers will read most JavaScript, but not until Version 4 of the two major browsers was JavaScript 1.2 available. Therefore, you really need your viewers to have at least Version 4 of either major browser for use with code from JavaScript 1.2. A guy in Outer Mongolia with an Internet connection has the same access to a new browser as a guy in Silicon Valley; all he has to do is to download and install either browser for free.

However, to get around the holdout who thinks that technology ended with his Version 2 Netscape Navigator, you can enter a simple set of semitags to mask the JavaScript. Because the older browsers don't know JavaScript from Sanskrit, they think that the code is text to be displayed on the page. To hide the JavaScript, you can place the container made up of `<!--` and `//-->` around the JavaScript code. For example, the following script segment is hidden from older browsers, and their parsers will skip over it:

```
<script language="JavaScript">
<!--
document.write("The old browsers cannot read this.")
//-->
```

Rarely do you find anyone still using browsers older than Version 4, and unless you want to degrade your JavaScript to an earlier version, you can include the masking container. However, at this point in browser development, it might be wiser to let the visitor know that her browser could use an upgrade by allowing JavaScript to appear on the screen. (A few cantankerous designers even use notes telling the viewer to upgrade his browser or get lost!)

Some designers attempt to write JavaScript with different sets of code for users with very old browsers by using browser-detection scripts written in JavaScript. In that way, users with older browsers can see some web page. In terms of cost benefits, having alternative sets of code for different browsers, different versions of browsers, and browsers for different platforms can become an onerous and expensive task. However, each designer/developer needs to decide her own willingness to have several different scripts for each page. With short scripts and a few pages, only a little extra work is required. However, with big sites and long sets of code, designers might find that they have to increase their time on the project and that they must charge their clients, or use the lowest common denominator of JavaScript.

Naming Rules and Conventions

All the rules for naming functions and variables discussed previously in the section on case sensitivity apply. If you name your variables or functions beginning with any letter in either uppercase or lowercase or with an underscore or dollar sign, the name is considered legitimate. For the most part, in naming variables, avoid using dollar signs as the first character because they can be confusing to Basic programmers, who might associate the dollar sign with a string variable, or PHP programmers, who begin all variables with a dollar sign.

You cannot have spaces in the names that you use for variables or functions. Many programmers use uppercase letters or underscores to make two words without a space. For example, the following are legitimate variable names where a space might be used in standard writing:

```
Bill_Sanders= "JavaScript guy";
BillSanders= "JavaScript guy";
Free$money="www.congame.html";
```

Whether to use an underscore or a cap to begin a new word in a variable or function name is your own preference.

Reserved Words

Reserved words are those in JavaScript that have been reserved for statements and built-in functions. If you use a reserved word for a variable or function name, JavaScript might execute the statement or function for the reserved word and not your variable or function. For example, if you use the reserved `break` for a variable name, JavaScript might attempt to jump out of a loop. Table 2.1 shows a list of current and future JavaScript reserved words.

TABLE 2.1 Reserved Words

abstract	final	public
boolean	finally	return
break	float	short
byte	for	static
case	function	super
catch	goto	switch
char	if	synchronized
class	implements	this
const	import	throw
continue	in	throws
debugger	instanceof	transient
default	int	true
delete	interface	try
do	long	typeof
double	native	var
else	new	void
enum	null	volatile
export	package	while
extends	private	with
false	protected	

You should also try to avoid words that you will find used in statements, methods, and attributes. For example, `name` and `value` are attributes of forms and should be avoided. As you learn more JavaScript, try to avoid the new words that you learn as names for variables and functions.

A Weakly Typed Language Means That JavaScript Is Smart

One last characteristic of JavaScript needs to be discussed before going on to the next chapter. JavaScript is considered a "weakly typed" or "untyped" language. The *type* in question here are the *data types*—nothing to do with typing from your keyboard. For programmers coming from C++ or Java, two strongly typed languages,

this means that JavaScript will figure out what type of data you have and make the necessary adjustments so that you don't have to redefine your different types of data. Designers new to programming will welcome a weakly typed language because it will save time in learning several different conversion steps and data type declarations.

For example, suppose that you're setting up an e-business site with lots of financial figuring that you want to slap a dollar sign on when all of the calculations are finished. Performing calculations involving money requires floating-point numbers. However, as soon as a dollar sign is added to the number, it becomes a string variable. In strongly typed languages, you need to convert the floating-point number to a string and then concatenate it with the dollar sign. Consider the following example using JavaScript to add five different items and placing the whole thing into a string.

WeaklyType.html

```
<html>
<head>
<script language="JavaScript">
var apples=1.43;
var oranges=2.33;
var pears=4.32;
var tax=.04;
var shipping=2.75;
var subtotal=apples + oranges + pears;
var total=subtotal + (subtotal * tax) + shipping;
var message="Your total is $";
var deliver= message + total +".";
document.write(deliver);
</script>
</head>
<body bgcolor="indianred">
</body>
</html>
```

Of course, when the calculations are all complete, you could drop the decimal points to two, as was done in the example script `VarFuncCase.html` in the previous section. However, the point is that no type definitions were required. Strings and numbers were merrily mixed with no cause for concern. Although the term "weakly typed" might imply some deficiency with JavaScript, the term actually means that JavaScript is smart enough to do the work of determining what type of data any given variable should be. In learning about data types and variables in the next chapter, you will be very grateful that JavaScript (and not you) does most of the work in figuring out data types.

Summary

The purpose of this short chapter has been to get you off on the right foot when you begin writing your own scripts. The most frustrating experience in learning how to program a new language is debugging it when the outcome is not the one expected. By paying attention to certain details and learning a set of rules for writing JavaScript, not only are you less likely to run into bugs when you create a page that includes JavaScript, but you are more likely to locate and correct them when they do pop up.