

Math, Number, and Boolean Objects

The introduction to data types and values in Chapter 6's tutorial scratched the surface of JavaScript's numeric and Boolean powers. In this chapter, you look more closely at JavaScript's way of working with numbers and Boolean data.

Math often frightens away budding programmers, but as you've probably seen so far in this book, you don't really have to be a math genius to program in JavaScript. The powers described in this chapter are here when you need them — if you need them. So if math was not your strong suit in school, don't freak out over the terminology here.

An important point to remember about the objects described in this chapter is that like string values and string objects, numbers and Booleans are both values and objects. Fortunately for script writers, the differentiation is rarely, if ever, a factor unless you get into some very sophisticated programming. To those who actually write the JavaScript interpreters inside the browsers we use, the distinctions are vital. As the language evolves, its behavior is increasingly formalized to a point at which the core language attributes (of which strings and numbers are a part) have been documented and published as an industry standard (ECMA-262). That standard serves as a guideline for all organizations that build JavaScript interpreters into their products. These folks, in turn, make the scripter's life easier by generally allowing us to treat a number or Boolean as a value or object as we please.

For most scripters, the information about numeric data types and conversions as well as the Math object are important to know. Other details in this chapter about the number and Boolean objects are presented primarily for completeness, since their direct powers are almost never used in day-to-day scripting of Web applications.

27 CHAPTER



In This Chapter

Advanced math operations

Number base conversions

Working with integers and floating-point numbers



Numbers in JavaScript

More powerful programming languages have many different kinds of numbers, each related to the amount of memory they occupy in the computer. Managing all these different types may be fun for some, but it gets in the way of quick scripting. A JavaScript number has only two possibilities. It can be an integer or a floating-point value. An *integer* is any whole number within a humongous range that does not have any fractional part. Integers never contain a decimal point in their representation. *Floating-point numbers* in JavaScript spread across the same range, but they are represented with a decimal point and some fractional value. If you are an experienced programmer, refer to the discussion about the number object later in this chapter to see how the JavaScript number type lines up with numeric data types you use in other programming environments.

Integers and floating-point numbers

Deep inside a computer, the microprocessor has an easier time performing math on integer values as compared to any number with a decimal value tacked on it, which requires the microprocessor to go through extra work to add even two such floating-point numbers. We, as scripters, are unfortunately saddled with this historical baggage and must therefore be conscious of the type of number used in certain calculations.

Most internal values generated by JavaScript, such as index values and length properties, consist of integers. Floating-point numbers usually come into play as the result of the division of numeric values, special values such as pi, and human-entered values such as dollars and cents. Fortunately, JavaScript is forgiving if you try to perform math operations on mixed numeric data types. Notice how the following examples resolve to the appropriate data type:

```
3 + 4 = 7 // integer result
3 + 4.1 = 7.1 // floating-point result
3.9 + 4.1 = 8 // integer result
```

Of the three examples, perhaps only the last result may be unexpected. When two floating-point numbers yield a whole number, the result is rendered as an integer.

When dealing with floating-point numbers, be aware that not all browser versions return the precise same value down to the last digit to the right of the decimal. For example, the following listing shows the result of 8/9 as calculated by numerous scriptable browsers (all Windows 95) and converted for string display:

Navigator 2	0.8888888888888884
Navigator 3	.8888888888888888
Navigator 4	.8888888888888888
Internet Explorer 3	0.888888888888889
Internet Explorer 4	0.888888888888888

It is clear from this display that you don't want to use floating-point math in JavaScript browsers to plan spaceflight trajectories. But it also means that for everyday math, you need to be cognizant of floating-point errors that accrue in PC arithmetic.

In Navigator, JavaScript relies on the operating system's floating-point math for its own math. Operating systems that offer accuracy to as many places to the right of the decimal as JavaScript displays are exceedingly rare. As you can detect from the preceding table, the more modern versions of browsers from Netscape and Microsoft are in agreement about how many digits to display and how to perform internal rounding for this display. That's good for the math, but not particularly helpful when you need to display numbers in a specific format.

JavaScript does not currently offer any built-in facilities for formatting the results of floating-point arithmetic. Listing 27-1 demonstrates a generic formatting routine for positive values, plus a specific call that turns a value into a dollar value. Remove the comments, and the routine is fairly compact.

Listing 27-1: A Generic Number Formatting Routine

```
<HTML>
<HEAD>
<TITLE>Number Formatting</TITLE>
<SCRIPT LANGUAGE="JavaScript">
// generic positive number decimal formatting function
function format (expr, decplaces) {
    // raise incoming value by power of 10 times the
    // number of decimal places; round to an integer; convert to
string
    var str = "" + Math.round (eval(expr) * Math.pow(10,decplaces))
    // pad small value strings with zeros to the left of rounded
number
    while (str.length <= decplaces) {
        str = "0" + str
    }
    // establish location of decimal point
    var decpoint = str.length - decplaces
    // assemble final result from: (a) the string up to the position of
    // the decimal point; (b) the decimal point; and (c) the balance
    // of the string. Return finished product.
    return str.substring(0,decpoint) + "." +
str.substring(decpoint,str.length);
}
// turn incoming expression into a dollar value
function dollarize (expr) {
    return "$" + format(expr,2)
}
</SCRIPT>
</HEAD>
<BODY>
<H1>How to Make Money</H1>
<FORM>
```

(continued)

Listing 27-1 (continued)

```
Enter a positive floating point value or arithmetic expression to be
converted to a currency format:<P>
<INPUT TYPE="text" NAME="entry" VALUE="1/3">
<INPUT TYPE="button" VALUE=">Dollars and Cents">
onClick="this.form.result.value=dollarize(this.form.entry.value)">
<INPUT TYPE="text" NAME="result">
</FORM>
</BODY>
</HTML>
```

This routine may seem like a great deal of work, but it's essential if your script relies on floating-point values and specific formatting.

Floating-point numbers can also be entered with exponents. An exponent is signified by the letter “e” (upper- or lowercase), followed by a sign (+ or -) and the exponent value. Here are examples of floating-point values expressed as exponents:

```
1e6 // 1,000,000 (the "+" symbol is optional on positive exponents)
1e-4 // 0.0001 (plus some error further to the right of the decimal)
-4e-3 // -0.004
```

For values between 1e-5 and 1e15, JavaScript renders numbers without exponents. All other values outside these bounds come back with exponential notation.

Hexadecimal and octal integers

JavaScript allows you to work with values in decimal (base-10), hexadecimal (base-16), and octal (base-8) formats. You only have a few rules to follow when dealing with any of these values.

Decimal values cannot begin with a leading 0. Therefore, if your page asks users to enter decimal values that may begin with a 0, your script must strip those zeroes from the input string or use the number parsing global functions (described in the next section) before performing any math on the values.

Hexadecimal integer values are expressed with a leading “0x” or “0X”. That’s a zero, not the letter “o”. The A through F values can appear in upper- or lowercase, as you prefer. Here are some hex values:

```
0X2B
0X1a
0xcc
```

Don’t confuse the hex values used in arithmetic with the hexadecimal values used in color property specifications for Web documents. Those values are expressed in a special *hexadecimal triplet* format, which begins with a crosshatch symbol followed by the three hex values bunched together (such as #c0c0c0).

Octal values are represented by a leading 0 followed by any digits between 0 and 7. Octal values consist only of integers.

You are free to mix and match base values in arithmetic expressions, but JavaScript renders all results in decimal form. For conversions to other number bases, you have to use a user-defined function in your script. Listing 27-2, for example, is a function that converts any decimal value from 0 to 255 into a JavaScript hexadecimal value.

Listing 27-2: Decimal-to-Hexadecimal Converter Function

```
function toHex(dec) {  
    hexChars = "0123456789ABCDEF"  
    if (dec > 255) {  
        return null  
    }  
    var i = dec % 16  
    var j = (dec - i) / 16  
    result = "0X"  
    result += hexChars.charAt(j)  
    result += hexChars.charAt(i)  
    return result  
}
```

The `toHex()` conversion function assumes that the value passed to the function is a decimal integer.

Converting strings to numbers

What has been missing so far from this discussion is a way to convert a number represented as a string to a number with which the JavaScript arithmetic operators can work. Before you get too concerned about this, be aware that most JavaScript operators and math methods gladly accept string representations of numbers and handle them without complaint. You will run into the data type incompatibilities most frequently when you are trying to accomplish addition with the `+` operator, but a string representation gets in the way. Also be aware that if you are performing math operations on values extracted from form text boxes, those object value properties are strings. Therefore, in many cases, those values need to be converted to values of the number type for math operations.

Conversion to numbers requires one of two JavaScript functions:

```
parseInt(string [,radix])  
parseFloat(string [,radix])
```

These functions were inspired by the Java language and are used here for compatibility reasons. The term *parsing* has many implied meanings in programming. One meaning is the same as *extracting*. The `parseInt()` function returns whatever integer value it can extract from the string passed to it; the `parseFloat()` function returns the floating-point number that can be extracted from the string. Here are some examples and their resulting values:

```
parseInt("42")           // result = 42  
parseInt("42.33")        // result = 42
```

```
parseFloat("42.33")    // result = 42.33
parseFloat("42")        // result = 42
parseFloat("fred")      // result = NaN
```

Because the `parseFloat()` function can also work with an integer and return an integer value, you may prefer using this function in scripts that have to deal with either kind of number, depending on the string entered into a text field by a user.

An optional second parameter to both functions lets you specify the base of the number represented by the string. This comes in particularly handy when you need a decimal number from a string that starts with one or more zeros. Normally the leading zero indicates an octal value. But if you force the conversion to recognize the string value as a decimal, it is converted the way you'd expect:

```
parseInt("010")    // result = 8
parseInt("010",10) // result = 10
parseInt("F2")     // result = NaN
parseInt("F2", 16) // result = 242
```

Use these functions wherever you need the integer or floating-point value. For example

```
var result = 3 + parseInt("3") // result = 6
var ageVal = parseInt(document.forms[0].age.value)
```

The latter technique ensures that the string value of this property is converted to a number (although I'd probably do more data validation — see Chapter 37 — before trying any math on a user-entered value).

Converting numbers to strings

If you attempt to pass a numeric data type value to many of the string methods discussed in Chapter 26, JavaScript complains. Therefore, you should convert any number to a string before you can, for example, find out how many digits make up a number.

You have two ways to force conversion from any numeric value to a string. The old-fashioned way is to precede the number with an empty string and the concatenation operator. For example, assume that a variable named `dollars` contains the integer value of 2500. To use the string object's `length` property (discussed later in this chapter) to find out how many digits the number is, use this construction:

```
("" + dollars).length // result = 4
```

The parentheses force JavaScript to evaluate the concatenation before attempting to extract the `length` property.

A more elegant way is to use the `toString()` method. Construct such statements as you would to invoke any object's method. For example, to convert the `dollars` variable value, shown earlier, to a string, the statement is

```
dollars.toString()    // result = "2500"
```

This method has one added power (new from Navigator 3): You can specify a number base for the string representation of the number. Called the *radix*, the base

number is added as a parameter to the method name. Here is an example of creating a numeric value for conversion to its hexadecimal equivalent as a string:

```
var x = 30
var y = x.toString(16) // result = "1e"
```

Use a parameter of 2 for binary results; 8 for octal. The default is base 10. Be careful not to confuse these conversions with true numeric conversions. Results from the `toString()` method cannot be used as numeric operands in other statements.

When a number isn't a number

In a couple of examples in the previous section, you probably noticed that the result of some operations was a value named `NaN`. That value is not a string, but rather a special value that stands for Not a Number. For example, if you try to convert a string "joe" to an integer with `parseFloat()`, the function cannot possibly complete the operation. It reports back that the source string, when converted, is not a number.

When you design an application that requests user input or retrieves data from a server-side database, you frequently cannot be guaranteed that a value you need to be numeric is, or can be converted to, a number. If that's the case, you need a way to see if the value is a number before performing some math operation on it. JavaScript provides a special global function, `isNaN()`, that lets you test the "number-ness" of a value. The function returns `true` if the value is not a number and `false` if it is a number. For example, you can examine a form field that should be a number:

```
var ageEntry = parseInt(document.forms[0].age.value)
if (isNaN(ageEntry)) {
    alert("Try entering your age again.")
}
```

Note

`NaN` and `isNaN()` were implemented in Navigator 2 only on UNIX versions. By Navigator 3, the value and function were on all platforms. Neither item is part of Internet Explorer 3, but are both available in Internet Explorer 4.

Math Object

Whenever you need to perform math that is more demanding than simple arithmetic, look through the list of Math object methods for the solution.

Syntax

Accessing select object properties and methods:

```
Math.property
Math.method(value [, value])
```

About this object

In addition to the typical arithmetic operations (covered in detail in Chapter 32), JavaScript includes more advanced mathematical powers that are accessed in a way that may seem odd to you if you have not programmed in true object-

oriented environments before. Although most arithmetic takes place on the fly (such as `var result = 2 + 2`), the rest requires use of the JavaScript internal Math object (that's with a capital "M"). The Math object brings with it several properties (which behave like some other languages' constants) and many methods (which behave like some other languages' math functions).

The way you use the Math object in statements is the same way you use any JavaScript object: You create a reference beginning with the Math object's name (Math), a period, and the name of the property or method you need:

```
Math.property | method([parameter]...[,parameter])
```

Property references return the built-in values (things such as pi); method references require one or more values to be sent as parameters of the method and then return the result of the method after it performs its operation on the parameter values.

Properties

JavaScript Math object properties represent a number of valuable constant values in math. Table 27-1 best shows you those methods and their values as displayed to 16 decimal places.

Table 27-1
JavaScript Math Properties

<i>Property</i>	<i>Value</i>	<i>Description</i>
Math.E	2.718281828459045091	Euler's constant
Math.LN2	0.6931471805599452862	Natural log of 2
Math.LN10	2.302585092994045901	Natural log of 10
Math.LOG2E	1.442695040888963387	Log base-2 of E
Math.LOG10E	0.4342944819032518167	Log base-10 of E
Math.PI	3.141592653589793116	p
Math.SQRT1_2	0.7071067811865475727	Square root of 0.5
Math.SQRT2	1.414213562373095145	Square root of 2

Because these property expressions return their constant values, you use them in your regular arithmetic expressions. For example, to obtain the circumference of a circle whose diameter is in variable `d`, you use this statement:

```
circumference = d * Math.PI
```

Perhaps the most common mistakes scripters make with these properties are failing to capitalize the Math object name or observing the case-sensitivity of property names.

Methods

Methods make up the balance of JavaScript Math object powers. With the exception of the `Math.random()` method, all Math object methods take one or more values as parameters. Typical trigonometric methods operate on the single values passed as parameters; others determine which of the numbers passed along are the highest or lowest of the group. The `Math.random()` method takes no parameters but returns a randomized, floating-point value between 0 and 1 (but note that the method does not work on Windows or Macintosh versions of Navigator 2). Table 27-2 lists all the Math object methods with their syntax and descriptions of the values they return.

Table 27-2 Math Object Methods	
<i>Method Syntax</i>	<i>Returns</i>
<code>Math.abs(val)</code>	Absolute value of <i>val</i>
<code>Math.acos(val)</code>	Arc cosine (in radians) of <i>val</i>
<code>Math.asin(val)</code>	Arc sine (in radians) of <i>val</i>
<code>Math.atan(val)</code>	Arc tangent (in radians) of <i>val</i>
<code>Math.atan2(val1, val2)</code>	Angle of polar coordinates <i>x</i> and <i>y</i>
<code>Math.ceil(val)</code>	Next integer greater than or equal to <i>val</i>
<code>Math.cos(val)</code>	Cosine of <i>val</i>
<code>Math.exp(val)</code>	Euler's constant to the power of <i>val</i>
<code>Math.floor(val)</code>	Next integer less than or equal to <i>val</i>
<code>Math.log(val)</code>	Natural logarithm (base e) of <i>val</i>
<code>Math.max(val1, val2)</code>	The greater of <i>val1</i> or <i>val2</i>
<code>Math.min(val1, val2)</code>	The lesser of <i>val1</i> or <i>val2</i>
<code>Math.pow(val1, val2)</code>	<i>Val1</i> to the <i>val2</i> power
<code>Math.random()</code>	Random number between 0 and 1
<code>Math.round(val)</code>	N+1 when <i>val</i> >= n.5; otherwise N
<code>Math.sin(val)</code>	Sine (in radians) of <i>val</i>
<code>Math.sqrt(val)</code>	Square root of <i>val</i>
<code>Math.tan(val)</code>	Tangent (in radians) of <i>val</i>

HTML is not exactly a graphic artist's dream environment, so using trig functions to obtain a series of values for HTML-generated charting is not a hot JavaScript prospect. But in the future, as users communicate with Java applets that

are better at graphical tasks, you may want to try JavaScript for some data generation using these advanced functions — sending the results to the Java applet for charting. For scripters who were not trained in programming, math is often a major stumbling block. But as you've seen so far, you can accomplish a great deal with JavaScript by using simple arithmetic and a little bit of logic, leaving the heavy-duty math for those who love it.

Creating random numbers

The `Math.random()` method returns a floating-point value between 0 and 1. If you are designing a script to act like a card game, you need random integers between 1 and 52; for dice, the range is 1 to 6 per die. To generate a random integer between zero and any top value, use the following formula:

```
Math.round(Math.random() * n)
```

where *n* is the top number. To generate random numbers between a different range use this formula:

```
Math.round(Math.random() * n) + m
```

where *m* is the lowest possible integer value of the range and *n* equals the top number of the range minus *m*. In other words *n+m* should add up to the highest number of the range you want. For the dice game, the formula for each die would be

```
newDieValue = Math.round(Math.random() * 5) + 1
```

Math object shortcut

In Chapter 31, I show you more details about a JavaScript construction that lets you simplify the way you address Math object properties and methods when you have a bunch of them in statements. The trick is using the `with` statement.

In a nutshell, the `with` statement tells JavaScript that the next group of statements (inside the braces) refer to a particular object. In the case of the Math object, the basic construction looks like this:

```
with (Math) {  
    [statements]  
}
```

For all intervening statements, you can omit the specific references to the Math object. Compare the long reference way of calculating the area of a circle (with a radius of six units)

```
result = Math.pow(6,2) * Math.PI
```

to the shortcut reference way:

```
with (Math) {  
    result = pow(6,2) * PI  
}
```

Though the latter occupies more lines of code, the object references are shorter and more natural when you're reading the code. For a longer series of calculations involving Math object properties and methods, the `with` construction saves

keystrokes and reduces the likelihood of a case-sensitive mistake with the object name in a reference. You can also include other full-object references within the `with` construction; JavaScript attempts to attach the object name only to those references lacking an object name.

Number Object

<i>Properties</i>	<i>Methods</i>	<i>Event Handler</i>
MAX_VALUE	toString()	(None)
MIN_VALUE		
NaN		
NEGATIVE_INFINITY		
POSITIVE_INFINITY		
prototype		

Syntax

Creating a number object:

```
var val = new Number(number)
```

Accessing number object properties:

```
Number.property | method
```

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility		✓	✓		✓	✓

About this object

The number object is rarely used, because for the most part, JavaScript satisfies day-to-day numeric needs with a plain number value. But the number object contains some information and power of value to serious programmers.

First on the docket are properties that define the ranges for numbers in the language. The largest number (in both Navigator and Internet Explorer) is 1.79E+308; the smallest number is 2.22E-308. Any number larger than the maximum is `POSITIVE_INFINITY`; any number smaller than the minimum is `NEGATIVE_INFINITY`. It will be a rare day on which you accidentally encounter these values.

More to the point of a JavaScript object, however, is the `prototype` property. In Chapter 26, I show you how to add a method to a string object's prototype such that every newly created object contains that method. The same goes for the

Number.prototype property. If you have a need to add common functionality to every number object, this is where to do it. This prototype facility is unique to objects and does not apply to plain number values. For experienced programmers who care about such matters, JavaScript number objects and values are defined internally as IEEE double-precision 64-bit values.

Boolean Object

<i>Property</i>	<i>Method</i>	<i>Event Handler</i>
prototype	toString()	(None)

Syntax

Creating a Boolean object:

```
var val = new Boolean(BooleanValue)
```

Accessing Boolean object properties:

```
Boolean.property | method
```

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility		✓	✓		✓	✓

About this object

You work with Boolean values a lot in JavaScript — especially as the result of conditional tests. Just as string values benefit from association with string objects and their properties and methods, so, too, do Boolean values receive aid from the Boolean object. For example, when you display a Boolean value in a text box, the “true” or “false” string is provided by the Boolean object’s `toString()` method, even though you don’t have to invoke it directly.

The only time you need to even think about a Boolean object is if you wish to attach some property or method to Boolean objects that you create with the `new Boolean()` constructor. Parameter values for the constructor include the string versions of the values, numbers (0 for false; any other integer for true), and expressions that evaluate to a Boolean value. Any such new Boolean object would be imbued with the new properties or methods you have added to the `prototype` property of the core Boolean object.

