JavaScript Data Types and Variables: Chapter 2 - Learning Javascript
by Shelley Powers

Variables in JavaScript are basically named buckets of data, a way of creating a reference to that data—regardless of whether the data is a string, number, boolean, array, or other object—so that you can access the same data again and again. More importantly, you can use variables to persist data from one process to another. For instance, your JavaScript application can store the value of a form element in a variable, and manipulate that value without having to actually manipulate the form element itself.

This excerpt is from **Learning JavaScript, Second Edition**. You'll learn everything from primitive data types to complex features, including JavaScript elements involved with Ajax and dynamic page effects. By the end of the book, you'll be able to work with even the most sophisticated libraries and web applications.

The variable's *data type* is the JavaScript scripting engine's interpretation of the type of data that variable is currently holding. A string variable holds a string; a number variable holds a number value, and so on. However, unlike many other languages, in JavaScript, the same variable can hold different types of data, all within the same application. This is a concept known by the terms *loose typing* and*dynamic typing*, both of which mean that a JavaScript variable can hold different data types at different times depending on context.

With a loosely typed language, you don't have to declare ahead of time that a variable will be a string or a number or a boolean, as the data type is actually determined while the application is being processed. If you start out with a string variable and then want to use it as a number, that's perfectly fine, as long as the string actually contains something that resembles a number and not something such as an email address. If you later want to treat it as a string again, that's fine, too.

The forgiving nature of loose typing can end up generating problems. If you try to add two numbers together, but the JavaScript engine interprets the variable holding one of them as a string data type, you end up with an odd string, rather than the sum you were expecting. Context is everything when it comes to variables and data types with JavaScript.

This chapter covers the JavaScript primitive data types of `string`, `boolean`, and `number`, as well as the built-in functions for modifying values of these types. In addition, we'll look at two special data types in JavaScript, `null` and `undefined`, toward the end of the chapter. Along the way, we'll explore escape sequences in strings and take a brief look at Unicode. The chapter also delves into the topic of variables, including what makes valid and meaningful variable identifiers.

Identifying Variables

JavaScript variables have an identifier, scope, and a specific data type. Because the language is loosely typed, the rest, as they say, is subject to change without notice.

Variables in JavaScript are much like those in any other language; you use them to hold values in such a way that the values can be explicitly accessed in different places in the code. Each has an identifier that is unique to the scope of use (more on this later), consisting of any combination of letters, digits, underscores, and dollar signs. An identifier doesn't have a required format, other than it must begin with a character, dollar sign, or underscore:

```
_variableidentifier __variableidentifier variableIdentifier $variable_identifier var-ident
```

Starting with JavaScript 1.5, you can also use Unicode letters (such as) and digits, as well as escape sequences (such as `\u0009`) in variable identifiers. The following are also valid variable identifiers for JavaScript:

```
_valid T\u0009
```

Use special characters with caution, though, as some tools such as debuggers may have difficulty with them.

JavaScript is case-sensitive, which means it treats upper- and lowercase characters differently. For instance, JavaScript sees the following two variable identifiers as separate variables:

```
stringVariable stringvariable
```

An additional restriction on variable identifiers is that they can't be a JavaScript keyword, a list of which appears in Table 2.1, "JavaScript keywords". Other keywords may be added over time, as new versions of JavaScript (well, technically, ECMAScript) are released.

Table 2.1. JavaScript keywords

break	else	new	var
case	finally	return	void
catch	for	switch	while
continue	function	this	with
default	if	throw	
delete	in	try	
do	instanceof	typeof	

Due to proposed extensions to the ECMA-262 specification, the words in Table 2.2, "ECMA-262 specification reserved words" are also considered reserved words and can't be used as variable identifiers.

Table 2.2. ECMA-262 specification reserved words

abstract	enum	int	short
boolean	export	interface	static
byte	extends	long	super
char	final	native	synchronized
class	float	package	throws
const	goto	private	transient
debugger	implements	protected	volatile
double	import	public	public

In addition to the ECMAScript reserved words, certain JavaScript-specific words implemented in most browsers are considered reserved by implementation. Many are based on the Browser Object Model (BOM)—for example, objects such as `document` and `window`, which were briefly introduced in Chapter 1, *Hello JavaScript!*. Though not a definitive list, Table 2.3, "Typical reserved words in browsers" includes the more common words.

Table 2.3. Typical reserved words in browsers

alert	eval	location	open
array	focus	math	outerHeight
blur	function	name	parent
boolean	history	navigator	parseFloat
date	image	number	regExp
document	isNaN	object	status
escape	length	onLoad	string

Naming Guidelines

Apart from the variable naming restrictions covered in the preceding section, you can use any identifier for variables and functions within code, but several naming practices—many inherited from Java and other programming languages—can make the code easier to follow and maintain.

First, use meaningful words rather than something you've thrown together quickly. For instance, the variable identifier `interestRate` is more descriptive than the variable `intRt` or even `ir`. The latter two names are too cryptic and too difficult to understand, even within a given context.

You can also provide a data type clue as part of the name, using something such as the following, which is a string, holding a first name:

```
var strFirstName = "Shelley";
```

This type of naming convention—using the data type as part of the variable name—is known as *Hungarian notation*, and is especially popular in Windows development. As such, you'll most likely see it used within the older JScript applications created for Internet Explorer, but less often in more modern JavaScript development.

Another naming convention is to use a plural for collections of items:

```
var customerNames = new Array();
```

Typically, variables are not capitalized because capitalization is usually reserved for objects such as `String`:

```
var firstName = String("Shelley");
```

Reserving capitalization for objects makes them easier to differentiate from simple variables.

Functions and variables frequently start with lowercase letters, and incorporate a verb representing what the function is doing. It's pretty easy to guess what the following function is doing:

```
function validateNameInRegister(firstName,lastName) ...
```

Many times, variables and functions have one or more words concatenated into a unique identifier, following a format popularized in other languages and frequently referred to as *CamelCase* because of the up-down nature of the name, like a camel's humps:

```
validateName firstName
```

The CamelCase naming format makes the variable much more readable, though dashes or underscores between the variable "words" work as well:

```
validate-name first_name
```

The newer JavaScript libraries invariably use CamelCase notation, which I also prefer for my own applications.

Though you can use a dollar sign, number, or underscore to begin a variable, your best bet is to start with a letter. Unnecessary use of unexpected characters in variable names can make the code harder to read and follow, especially for newer JavaScript developers. However, if you've looked at some of the newer JavaScript libraries and examples, you might have noticed some odd-looking variable names. The popular Ajax-based Prototype JavaScript library is a strong influence in this regard—so much so that I think of the rise of new naming conventions as the "Prototype effect."

The following is an example of this effect:

```
var _break = someval;
```

The underscore is used in these libraries to signal a variable that's an object's private data member, a concept I'll cover in Chapter 13, *Creating Custom JavaScript Objects*. Another interesting naming variation that Prototype has introduced is the following, which uses the dollar sign (`$`) to name a function that returns a reference to a page element:

```
$('test').invokeSomeMethod();
```

The use of the underscore or dollar sign doesn't change the behavior of the variable, even though such usage is relatively new. It's just another way of naming something.

Note

You can find the Prototype JavaScript library at http://www.prototypejs.org/.

Aside from the few JavaScript naming restrictions, nothing is mandatory or magical about the naming conventions I've outlined. They help to make JavaScript easier to read and debug.

Primitive Types

JavaScript is a trim language, with just enough functionality to do the job—no more, no less. However, as I've said before, it is a confusing language in some respects.

For instance, JavaScript has just three primitive data types: `string`, `numeric`, and `boolean`. Each is differentiated from the others by the type of value it contains: string, numeric, and boolean, respectively. However, JavaScript also has built-in objects known as`String`, `Number`, and `Boolean`. These would seem to be very different from each other: the first three are types of primitive values, whereas the latter three are objects, each one with its own built-in properties and methods.

In actuality, the two are connected. The `String` object *wraps* the string primitive—just as the `Number` and `Boolean` objects wrap their individual primitive types—when using the primitive type like an object. When you create a simple string variable in JavaScript, and then use one of the `String` methods, JavaScript implicitly wraps the string primitive in a `String` object, processes the `String` object property or method call, and then discards the object. In the following code snippet, when the method `toUpperCase` is called on`firstName`, an object is created to wrap the string and then process the method call before the object is discarded:

```
var firstName = "Shelley"; var cappedName = firstName.toUpperCase();
```

For all intents and purposes, `firstName` looks like an object, and it acts like an object when calling `toUpperCase`, but it is a primitive. If I call another `String` object method, the same thing will happen again: a `String` object is created and then wraps the primitive, processes the method call, and discards the object. As you can imagine, if you're going to be treating a string like an object, you'd be better off creating it as an object. At the same time, if all you need is a simple string to print a message or hold a value, you don't need all the functionality that accompanies an object, so creating a string primitive is the better option.

Rather than continue to mix primitive data types and objects in a confusing mishmash that wanders from primitive to object and back again, in the next three sections, we'll look at each of the primitive data types—how they're created and manipulated, and how you can convert values of one type to other type. In Chapter 4, *The JavaScript Objects*, I'll cover the data objects, their methods, and their properties.

Note

When I use the word *wrap* in the book, I'm talking about an object that typically encloses a simpler item, such as a`String` object wrapping a string primitive.

The String Data Type

Because JavaScript is a loosely typed language, nothing differentiates a string variable from a variable that's a number or a boolean, other than the literal value assigned to the string variable when it's initialized and the context of its use.

A string literal is a sequence of characters delimited by single or double quotes:

```
var strString = "This is a string"; var anotherString= 'But this is also a string';
```

No rule states which type of quote you use, except that the ending quote character must be the same as the beginning one. You can include any variation of characters in the string:

```
var thirdString = "This is 1 string."; var stringFour = "This is--another string.";
var stringAsNumber = "543";
```

The last example of a string contains a number, but because it's surrounded by quotes, JavaScript creates the variable as a string.

A string can also include quotes. You can use single and double quotes interchangeably if you need to include a quote within a quoted string. All you have to do is be consistent—if the string contains a single quote, use double quotes around the string; the same is true with the double quote. For example:

```
var string_value = "This is a 'string' with a quote."
```

or:

```
var string_value = 'This is a "string" with a quote.'
```

The empty string is a special case: you'd commonly use it to initialize a string variable when it's defined. The following are examples of empty strings:

```
var string_value = ''; var anotherStringValue = "";
```

Which quote character you use makes no difference to the JavaScript engine. What's more important is to use one or the other consistently in your applications.

String Escape Sequences

Not all characters are treated equally within a string in JavaScript. A string can also contain an *escape sequence*, such as `\n` for the end-of-line terminator. An escape sequence is a pattern in which certain characters are encoded in certain ways in order to include them within a string.

The following snippet of code assigns a string literal containing a line-terminator escape sequence to a variable. When the string is used in a dialog window, the escape sequence, `\n`, is interpreted literally, and a newline is published:

```
var string_value = "This is the first line\nThis is the second line";
```

This results in:

```
This is the first line This is the second line
```

You can also use the backslash to denote that the quote in the string is meant to be taken as a literal character, not as an end-of-string terminator:

```
var string_value = "This is a \"string\" with a quote."
```

By using the backslash with quotes, you can include single and double quotes within a string.

To include a backslash in a string, use two backslashes in a row:

```
var string_value = "This is a \\string\\ with a backslash."
```

The result of this line of code is a string with two backslashes, one on each side of the word "string".

You can also include Unicode characters in a string by preceding the four-digit hexadecimal value of the character with `\u`. For instance, the following outputs the Chinese (simplified) ideogram for "love":

```
document.writeln("\u7231");
```

What displays is somewhat browser-dependent; however, most of the more commonly used browsers now have adequate Unicode support.

Note

You can learn more about Unicode, and access relevant charts, at http://www.unicode.org/.

String Encoding

Using the backslash to escape characters is helpful when you're using ASCII characters that are normally control characters within a string. However, the backslash can't do anything with characters that are not ASCII; nor can it do anything if you want to make an entire string safe for HTML processing, which is necessary for Ajax-based applications (I'll touch on this at the end of the book).

You use the `encodeURI` and `encodeURIComponent` methods to escape, or more properly, to *encode* entire strings, converting ASCII and non-ASCII characters to URI encoding, which you can use in links and Ajax applications.

Note

URI stands for Uniform Resource Identifier, an example of which is a web page URL. An example of URI encoding is ISO Latin-1 (also known as ISO 8859-1).

The `encodeURI` makes an assumption that the string is a URI such as "http://oreilly.com", and reserves the following characters:

```
; , / ? : @ & = + $
```

Alphanumeric characters and the following punctuation are also not encoded:

```
- _ . ! ~ * ' ( )
```

The page fragment symbol (#) is also not encoded.

The `encodeURIComponent`, however, encodes all characters except the alphanumeric and punctuation characters listed earlier. It assumes that the string being encoded is being used as a parameter to a URI, and therefore characters that are normally part of a URI, such as the following, are encoded:

```
# & + =
```

Both functions also have their opposite member: `decodeURI`, to decode the `encodeURI` encoded string, and `decodeURIComponent`, to decode the `encodeURIComponent` string.

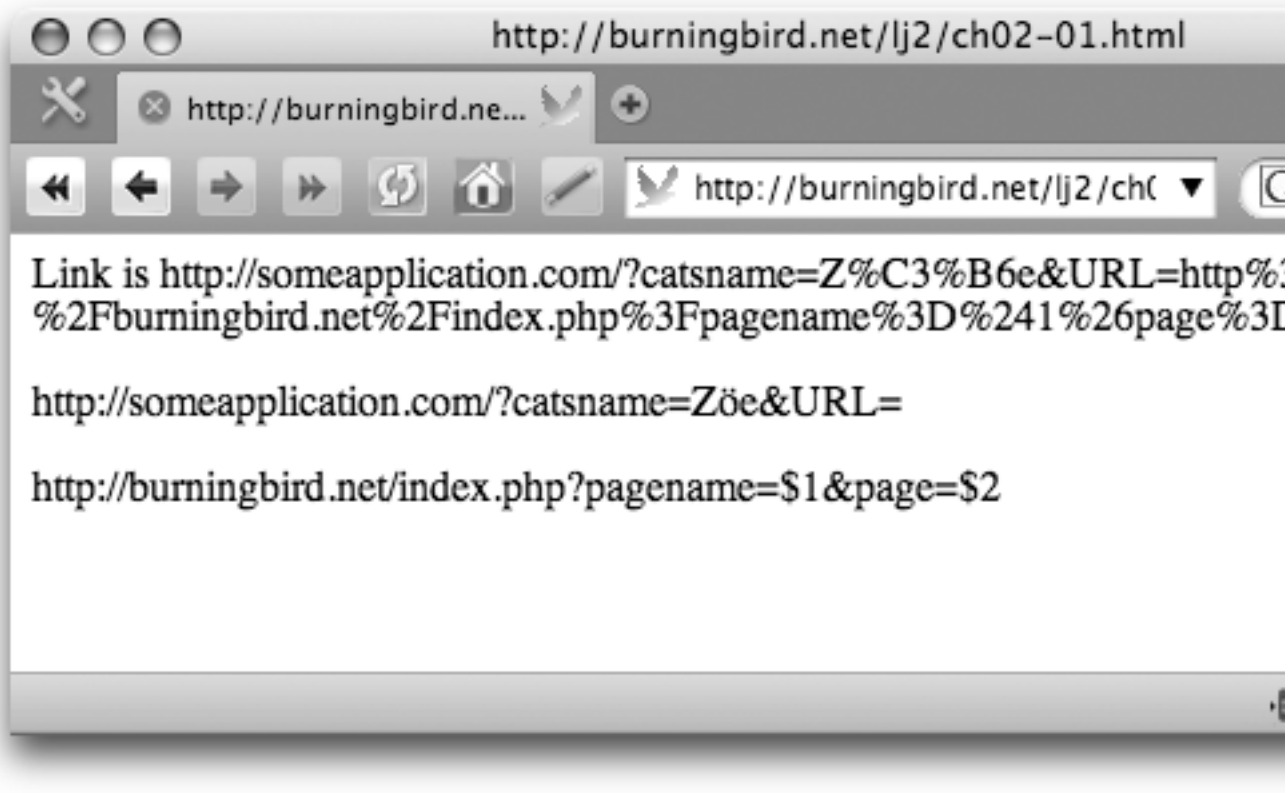
Example 2.1, "URI encoding two strings using the JavaScript encodeURI and encodeURIComponent methods" shows a web page that uses all four functions to encode and then decode two strings, all of which are then printed out to the current web page using `document.writeln`.

Example 2.1. URI encoding two strings using the JavaScript encodeURI and encodeURIComponent methods

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd"> <html
xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"> <head> <title>URI Encoding</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" /> <script
type="text/javascript"> //<![CDATA[  function encodeStrings() {     var sOne =
encodeURIComponent("http://burningbird.net/index.php?pagename=$1&page=$2");     var
sTwo = encodeURI("http://someapplication.com/?catsname=Z e&URL=");     var sOutput =
"<p>Link is " + sTwo + sOne + "</p>";     document.write(sOutput);      var
sOneDecoded = decodeURI(sTwo);     var sTwoDecoded = decodeURIComponent(sOne);
var sOutputDecoded = "<p>" + sOneDecoded + "</p><p>" + sTwoDecoded + "</p>";
document.write(sOutputDecoded);  } //]]> </script> </head> <body
onload="encodeStrings()">    <p></p> </body> </html>
```

Figure 2.1, "Result of encoding/decoding URI application" shows the resultant page.

Figure 2.1. Result of encoding/decoding URI application



These are all demonstrations of how to explicitly create a string variable, and variations of string literals that incorporate special characters. You can also convert the values within a specific variable from other data types, depending on the context.

Converting to Strings

You can convert other data types, such as numbers and booleans, to a string; typically, the scripting engine will do the conversion automatically, based on the context. As an example, if a numeric or boolean variable is passed to a function that expects a string, the value is implicitly converted to a string first, before the value is processed:

```
var num_value = 35.00; alert(num_value); // expects a string
```

In addition, when the plus sign (+) is used with two variables in an assignment statement, and one value is a string and the other anumber, the number is converted to a string and then the two strings are concatenated:

```
var num_value = 35.00; var string_value = "This is a number:" + num_value;
```

When the conversion from number to string occurs depends on when the JavaScript scripting engine encounters the string. For instance, if the string is the first in a sequence of values, all of the numbers that follow are treated as strings:

```
var strValue = "4" + 3 + 1; // becomes "431" var strValueTwo = 4 + 3 + "1"; // becomes
71
```

However, if you use operators other than +, the opposite type of conversion is applied—the string is converted to a number:

```
var firstResult = "35" - 3; // subtraction is applied, resulting in 32 var
secondResult = 30 / "3"; // division is applied, resulting in 10 var thirdResult = "3"
* 3; // multiplication is applied, resulting in 9
```

This implicit conversion with its dependency on both operator and position demonstrates more fully the danger of loose typing: the values you end up with can vary widely, depending on something as simple as where in the sequence of operations you introduce a new data type, or what type of operator you use.

Note

I cover the addition and other operators demonstrated in this chapter, as well as others available in JavaScript, more fully in Chapter 3, *Operators and Statements*.

Rather than depend fully on happenstance data type conversion, you can explicitly convert a variable to a string using the `String`global function. If the value being converted is a boolean, the resultant string is a text representation of the boolean value: `"true"` for true; `"false"` for false. For numbers, the string is, again, a string representation of the number, such as `"-123.06"` for –123.06, depending on the number of digits and the precision (the placement of the decimal point). A value of `NaN` (Not a Number, discussed later) returns `"NaN"`, whereas undefined or null variables will return `"undefined"` or `"null"`, respectively.

Table 2.4, "toString conversion table" shows the results of using `toString` on different data types.

Table 2.4. toString conversion table

Input	Result
`Undefined`	`"undefined"`
`Null`	`"null"`
`Boolean`	If true, then `"true"`; if false, then `"false"`
`Number`	The string representation of the number, or `NaN` if the variable holds this latter value
`String`	No conversion
`Object`	A string representation of the default representation of the object

The last item in the table describes the ECMAScript rule for the result of `toString` with an object. The representation is:

```
"[object "+className+"]"
```

Example 2.2, "Explicit and implicit string conversions" shows explicit string conversion on several different variables and objects. New number and boolean variables are created and initialized to data-type-specific values and then both are converted explicitly to strings using `String`. The example applies the same conversion to a variable that's created without an initial value, and one initialized to null. Lastly, it passes the document object to `String` and the resultant string is also printed out to the page.

Example 2.2. Explicit and implicit string conversions

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd"> <html
xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"> <head> <title>Implicit and
Explicit String Conversion</title> <meta http-equiv="Content-Type" content="text/html;
charset=utf-8" /> <script type="text/javascript"> //<![CDATA[  function
convertToString() {    var newNumber = 34.56;    var newBoolean = true;    var
nothing;    var newNull = null;     var strNumber = String(newNumber); var strBoolean
= String(newBoolean);    var strUndefined = String(nothing); var strNull =
String(newNull);     var strOutput = "<p>" + strNumber + " " + strBoolean + " " +
strUndefined + " " + strNull + "</p>";    document.writeln(strOutput);     var
strOutput2 = String(document);    document.writeln(strOutput2);  } //]]> </script>
</head> <body onload="convertToString()">    <p></p> </body> </html>
```

The output of this application varies among browsers. Firefox, Opera, IE, and Safari all output the first string in the same way:

```
34.56 true undefined null
```

However, only Opera and Firefox output the ECMAScript-specific object representation for `document`:

```
[object HTMLDocument]
```

IE outputs just `[object]` and Safari/WebKit doesn't output anything at all when using `String` with `document`.

The Boolean Data Type

The boolean data type has two possible values: `true` and `false`. They are not surrounded by quotes; in other words, `"false"` is not the same as `false`.

```
var isMarried = true; var hasChildren = false;
```

An implicit boolean value is assigned to variables of different types, and it depends on whether the variable is set and, if it is set, what value it has. For instance, the following conditional block will be processed if the variable, `testVariable`, implicitly converts to a boolean `true` value; it will not be processed if it is implicitly converted to a boolean `false`:

```
if (testVariable) {    ... }
```

We'll see later how to use boolean values to manage the flow of control for an application, but for now, Table 2.5, "toBoolean conversion table" shows how variables of different types would implicitly and explicitly convert to a boolean. I say "explicitly" because like the string data type, you can convert different data types into a boolean value explicitly using the `Boolean` function:

```
var someValue = 0; var someBool = Boolean(someValue) // evaluates to false
```

Table 2.5. toBoolean conversion table

Input	Result
`Undefined`	`false`
`Null`	`false`
`Boolean`	Value of `value`
`Number`	Value of `false` if `number` is 0 or NaN; otherwise, `true`
`String`	Value of `false` if `string` is empty; otherwise, `true`
`Object`	`true`

You can also use *double negation* (the negation operator, `!`, used twice) to explicitly convert a number or string to a boolean:

```
var strValue = "1"; var numValue = 0; var boolValue = !!strValue; // converts "1" to a
true boolValue = !!numValue; // converts 0 to false
```

The Number Data Type

Number data types in JavaScript are floating-point numbers, but they may or may not have a fractional component. If they don't have a decimal point or fractional component, they're treated as integers—base-10 whole numbers in a range of -2^{53} to 2^{53}.
The following are valid integers:
```
var negativeNumber = -1000; var zero = 0; var fourDigits = 2534;
```
The floating-point representation has a decimal, with a decimal component to the right. You also can represent the number as an exponent, using scientific notation. All of the following are valid floating-point numbers:
```
var someFloat = 0.3555 var anotherNumber = 144.006; var negDecimal = -2.3; var lastNum
= 19.5e-2 //which is equivalent to .195 var zeroDecimal = 12.0;
```
Though larger numbers are supported, some functions can work only with numbers in a range of –2e31 to 2e31 (–2,147,483,648 to 2,147,483,648); as such, you should limit your number use to this range.
Two special numbers exist: positive and negative infinity. In JavaScript, they are represented by `Infinity` and `-Infinity`, respectively. A positive infinity is returned whenever a math overflow occurs in a JavaScript application. A negative infinity is returned when a number occurs that is smaller than the minimum value supported in JavaScript.
In addition to base-10 representation, you can use octal and hexadecimal notation with JavaScript numbers, though octal is newer and may be confused for hexadecimal with older browsers. A hexadecimal number begins with a zero, followed by an `x`:
```
var firstHex = -0xCCFF;
```
An octal value begins with a zero, but there is no leading `x`:
```
var firstOct = 0526;
```
What's interesting with both the octal and decimal representations is that if you convert either to a string using the `String` function (covered earlier), the scripting engine first converts the number to its base-10 (decimal) representation and then converts the result to a string. So, a string conversion of `firstOct` is `"342"`, which is the string conversion of the decimal conversion of the octal number 0526.
In the preceding two sections, I demonstrated how to convert other types to a string or a boolean. In this section, I'm going to demonstrate two different functions you can use to convert a string to a number: `parseInt` and `parseFloat`. The `parseInt` functionreturns the integer portion of a number in a string, regardless of whether the string is formatted as an integer or as a floating-point number. The `parseFloat` function returns the floating-point value until a character that is not a sign (`+` or `-`), decimal, number, or exponent is reached.
Example 2.3, "Converting strings to numbers using either parseInt or parseFloat" passes three strings containing numeric values to either `parseInt` or `parseFloat`, and writes the values to the page.

Example 2.3. Converting strings to numbers using either parseInt or parseFloat

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd"> <html
xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"> <head> <title>Convert to
Number</title> <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript"> //<![CDATA[  function convertToNumber() {    var sNum
= "1.23e-2";    document.writeln("<p>" + parseFloat(sNum) + "</p>");
document.writeln("<p>" + parseInt(sNum) + "</p>");     var fValue = parseFloat("1.45
inch");    document.writeln("<p>" + fValue + "</p>");     var iValue = parseInt("-
33.50");    document.writeln("<p>" + iValue + "</p>");  } //]]> </script> </head>
<body onload="convertToNumber()">  <p></p> </body> </html>
```

The following is the output of the page using Firefox, Safari/WebKit, Opera, and IE8:
```
0.0123 1 1.45 -33
```
Notice with the resultant first value that the number is printed in decimal notation rather than the exponential notation of the original string value. The second printed value, which is the first string converted using `parseInt`, is truncated right at the decimal and before the exponent was applied to the number. Also, note that `parseInt` truncates the fractional component of the number in the fourth conversion result.
The third number conversion is the really interesting conversion. The `parseFloat` function grabbed the number from the string, `"1.45 inch"`, up to the first nonnumber-specific value—in this case, the space between "1.45" and "inch". The function then converted the string `"1.45"` to the float, which was printed.

Note

Of course, the numbers shown in Example 2.3, "Converting strings to numbers using either parseInt or parseFloat"are then converted back into a string for writing out to the page.

The `parseInt` function can convert a decimal to an octal or a hexadecimal and back again. A second parameter to the function, `base`, is equivalent to the number *radix*, and is 10 or base 10, by default. If any other base is specified in a range from 2 to 36, the string is interpreted accordingly. The following JavaScript:
```
var iValue = parseInt("266",16); document.writeln("<p>" + iValue + "</p>");  var
iValue = parseInt("55",8); document.writeln("<p>" + iValue + "</p>");
```
results in the two converted values being printed to the page:
```
550  45
```
In addition to `parseInt` and `parseFloat`, the `Number` function also converts other data types to numbers. The type returned after conversion is dependent on the representation: floating-point

strings return floating-point numbers; integer strings return integers.Table 2.6, "Conversion from other data types to numbers" shows the conversion to numbers from each type.

Table 2.6 Conversion from other data types to numbers

Input	Result
`Undefined`	`NaN`
`Null`	`0` (note that IE returns `NaN`)
`Boolean`	If `true`, the result is `1`; otherwise, `0` (note that IE returns `NaN`)
`Number`	Straight value
`String`	Integer or float, depending on representation
`Object`	`NaN`

In addition to converting strings to numbers, you can also test the value of a variable to see whether it's infinity through the `IsFinite`function. If the value is infinity or `NaN`, the function returns `false`; otherwise, it returns `true`.

Other functions work on numbers, but they're associated with the `Number` object, discussed in Chapter 4, *The JavaScript Objects*. For now, we'll continue to look at the primitive types with two special JavaScript types: `null` and `undefined`.

The null and undefined Variables

Nowhere in JavaScript is the line between literals, simple data types, and objects blurred more than it is when you're looking at two variables that represent nonexistence or incomplete existence: `null` and `undefined`.

A `null` variable is one that you have defined and to which you have assigned *null* as its value. The following is an example of a `null`variable:

```
var nullString = null;
```

If you have declared but not initialized the variable, it is considered undefined:

```
var undefString;
```

A variable is not null and is not undefined when you declare it and give it an initial value:

```
var sValue = "";
```

When you're using several JavaScript libraries and fairly complex code it's not unusual for a variable not to be set; if you try to use the variable in an expression, you can get an adverse result—usually a JavaScript error. One approach to testing variables if you're unsure of their state is to use the variable in a conditional test, such as the following:

```
if (sValue) ... // if not null and initialized, true; otherwise false
```

We'll look at conditional statements in Chapter 3, *Operators and Statements*, but for now, know that the expression consisting of just the variable `sValue` evaluates to `true` if `sValue` has been declared and initialized; otherwise, the result of the expression is `false`:

```
if (unknownVariable)// false, variable is not declared or assigned a value  if
(undefinedString) // false, as variable has not been given a value  if (nullString) //
variable has been defined and given a value, but that value is null and so the result
is false  if (sValue) // true if a variable is both defined and given a value
(including empty string)
```

Using the `null` keyword, you can specifically test to see whether a value is null:

```
if (sValue == null)
```

In JavaScript, a variable is undefined, even if it is declared, until it is initialized. A variable can be undeclared but initialized, in which case it is not null and is not undefined. However, in this instance, it's considered a global variable, and as discussed earlier, not specifically declaring variables with `var` causes problems more often than not.

Note

In some of the code snippets, comments that begin with `//` wrap to the next line because of the page width, not because the comments are actually multiple lines.

Though not related to existence, a third unique value is related to the type of a variable: `NaN`, or Not a Number. If a string or boolean variable cannot be coerced into a number, it's considered `NaN` and is treated accordingly:

```
var nValue = 1.0; if (nValue == 'one' ) // false, the second operand is NaN
```

You can specifically test whether a variable is `NaN` with the `isNaN` function:

```
if (isNaN(sValue)) // if string cannot be implicitly converted into number, return
true
```

By its very nature, a null value is `NaN`.

Note

Author and respected technologist Simon Willison gave an excellent talk at O'Reilly's 2006 ETech conference, titled "A (Re)-Introduction to JavaScript." You can view his slides at his website, http://simon.incutio.com/. The whole presentation is a very worthwhile read, but my favorite is the following line:

"In other words, zero, null, NaN, and the empty string are inherently `false`; everything else is inherently `true`."

For the most part, JavaScript developers create code in such a way that we know a variable is going to be defined ahead of time and/or given a value. In most instances, we don't explicitly test to see whether a variable is set, and if so, whether it's assigned a value.

However, when using large and complex JavaScript libraries, and applications that can incorporate web service responses, it becomes increasingly important to test variables that originate and/or are set outside our control. It is also increasingly important to be aware of how null and undefined variables behave when accessed in the application.

Constants: Named but Not Variables

Sometimes you'll want to define a value once, and then have it treated as a read-only value from that time forward. You use the keyword `const` to create a JavaScript constant:

```
const CURRENT_MONTH = 3.5;
```

The constant can be of any value, and because it can't be assigned or reassigned a value at a later time, it's initialized to its constant value when defined.

Just as with variables, a JavaScript constant has global and local scope. I use constants at a global level, primarily because they contain a value I want to be accessible (and unchanged) by a JavaScript block. Also, notice how the entire constant name is in uppercase. This isn't required, but it is a fairly standard naming convention, making constants easier to spot in code.

Test Your Knowledge: Quiz

1. Of the following identifiers, which are valid, which are not, and why?
   ```
   $someVariable _someVariable 1Variable some_variable som€variable function
   some*variable
   ```
2. Convert the following identifiers to CamelCase notation:
   ```
   var some_month; function theMonth // function to return current month current-
   month // a constant var summer_month; // an array of summer months MyLibrary-
   afunction // a function from a JavaScript package
   ```
3. Is the following string literal valid? If not, how would you fix it?
   ```
   var someString = 'Who once said, "Only two things are infinite, the universe
   and human stupidity, and I'm not sure about the former."'
   ```
4. Given a number, 432.54, what JavaScript function(s) returns the integer component of the number, and then finds the hexadecimal and octal conversions?
5. You're creating a JavaScript function in a library that other applications can use. A parameter, `someMonth`, is passed to the function. How would you determine whether it's null or undefined?

Test Your Knowledge: Answers

1. The following are valid:
   ```
   $someVariable _someVariable some_variable som€variable
   ```
   The following are not valid:
   ```
   1Variable does not start with a valid character function is a reserved word
   some*variable uses an invalid character
   ```
2. The identifiers are converted as follows:
   ```
   var someMonth function getCurrentMonth CURRENT_MONTH summerMonths
   myLibraryFunction
   ```
3. The string is not valid. To make it valid, either use single quotes only within the double quote, or escape the double quotes:
   ```
   var someString = "Who once said, 'Only two things are infinite, the universe
   and human stupidity, and I'm not sure about the former'";
   ```
   or:
   ```
   var someString = 'Who once said, "Only two things are infinite, the universe
   and human stupidity, and I\'m not sure about the former"';
   ```
   Don't forget to escape the apostrophes in contractions.
4. The following code would work:
   ```
   var fltNumber = 432.54; var intNumber = parseInt(fltNumber); var octNumber =
   intNumber.toString(8); var hexNumber = intNumber.toString(16);
   ```
5. This is a trick question. Passing a variable that's not declared or defined to a function or object method results in a JavaScript error, so your function will not have to test the parameter.
   Use the following to test to see whether a variable has been set elsewhere in code:
   ```
   if (a) {    ... }
   ```

If you enjoyed this excerpt, buy a copy of **Learning JavaScript, Second Edition**.