Chapter 4, Introducing JavaScript: Objects, Types and Variables

John M. Morrison

September 27, 2017

Contents

0 Introduction **1**

1 JavaScript's Type System: numbers **3**

1.1 What about all of my favourite math functions? 4

1.2 Oh crud! How do I find everything in the `Math` object? 5

2 The JavaScript Type System: Strings **6**

2.1 Can we get a chunk of a string? 7

3 The JavaScript Type System: Boolean **9**

4 Variables and Expressions **10**

5 Seeing JavaScript in a Web Page **13**

6 Please Excuse My Dear Aunt Sally **18**

7 More about Objects **19**

0 Introduction

So far, we have encountered the language HTML, which specifies the structure of a web page, and the language CSS, which specifies its appearance. Now we

come to the part that gives web pages behaviour, the programming language JavaScript.

As we said before, the browser is a virtual computer that understands three languages. It uses HTML to create the structure of a page, CSS to determine a page's appearance, and JavaScript is a programming language that gives pages behavior.

We are going to begin with some very basic ideas to get things started. For now, there are two items of importance to you. Objects are items stored in memory that can represent data or accomplish tasks. Variables are names we can attach to objects; these can be used as a means of communicating with an object to get it to execute some desired behavior, or to learn its properties.

So, let us begin by learning about a few fundamental kinds of objects.

A Word to the Wise The best way to read this chapter is to open the console in Firefox or Chrome; you will be shown how to do this in the next section. As you are reading this chapter *experiment*. Do not be afraid to make mistakes; if you do something nonsensical, the console will hiss at you. As you look at examples, try typing similar things into the console. Do some exploring; this is the best way to learn.

What is Node? Node is a JavaScript interpeter tht allows you to run JavaScript programs at the command line. It is freely available for Mac, UNIX and Windoze machines. You can obtain it at `https://nodejs.org/en/`. A how-to article on installing it in Windoze can be seen at `http://blog.teamtreehouse.com/install-node-js-npm-windows`. You can run a JavaScript file in node as follows. Create this file

```
console.log("Hello, World!");
```

In your UNIX terminal or your PowerShell window type

```
$ node hello.js
Hello, World!
```

If you can do this, you have installed Node successfully.

You can also use it interactvely. Just enter `node` at the command line. It will show you a prompt that looks like this `>`. You enter JavaScript and it replies. Here is an example.

```
$ node
> console.log("foo")
foo
```

```
undefined
>
```

To quit, enter control-D or `process.exit()`. Your original prompt will reappear. Node is a nice alternative to the browser's console, which you will also learn about.

1 JavaScript's Type System: numbers

Every object has a type, just as every living thing has a species. You can think of the type of an object as its species.

When we think of computing, we often think first of numbers. JavaScript has a number type. We can "talk" to JavaScript using both Firefox and Chrome. For both, click on the ≡ symbol in the upper right hand corner of the browser window, or right-click and choose "Inspect" from the menu. In Firefox, you will see a wrench icon labeled "Developer;" click on this. In Chrome select "More Tools" from the menu, then "Developer Tools."

You will see a window with tabs; click on Console. In Chrome, you will see a prompt that looks like this.

```
>
```

In Firefox, the prompt will look like this.

```
>>
```

At the prompt, type an expression such as `5*2`. Here is what you will see in Firefox.

```
>> 5*2
<- 10
```

Here we run some expressions in Chrome.

```
> 5*2
<- 10
> 5+2
<- 7
> 5/2
<- 2.5
> 5 - 2
<- 3
```

What we see here is that number objects have exactly the behavior you expect when combined with the four basic arithmetic operations. Can we do anything else with these?

Here is operation you never saw explicitly defined in your Mrs. Wormwood days, %.

```
> 5 % 2
<- 1
```

This operation is called *mod*. It takes the right-hand operand, divides it into the left, and computes the remainder. Since when we divide 5 by 2, we get a quotient of 2 and a remainder of 1, we have `5 % 2` evaluating to 1.

This operation was introduced by Mrs. Wormwood back in the '60s and the New Math days. Here is what they told elementary school kids. Suppose you are computing `41% 8`. To do this imagine you have a clock with the values 0-7 on its face instead of the usual 1-12. Put the 0 at the top position and number the numbers clockwise. Now move the hand ahead 41 "hours;" you will make 5 revolutions, then one additional click. If the hand started at 0, it will end at 1. This tells you that `41% 8` is 1.

Think about it for a moment. You have

$$41 = \text{number of numbers om the dial} * \text{number of revolutions of the dial} + 1.$$

Puzzler What does the fact that `365 % 7` evaluates to 1 have to do with your birthday?

Finally, let us see that numbers do know their type. Here is how we see that You can use the `typeof` operator to learn the type of any JavaScript object.

```
> typeof 5
"number"
```

1.1 What about all of my favourite math functions?

JavaScript has something called the *Math object* that serves these up. Using the `Math` object is simple. Look at these examples.

```
> Math.sqrt(5)
<- 2.23606797749979

> Math.pow(5,4)
<- 625
> Math.sin(Math.PI/2)
<- 1
```

To talk to the `Math` object, you type `Math.` then the function you wish to use. All of your favorites are present. Notice that the default angular unit for the trigonometric functions is radian measure. The item `Math.PI` is a symbolic constant for the value π so it is easy to use the conversion factors of $\pi/180$ and $180/\pi$ to convert between degrees and radians.

1.2 Oh crud! How do I find everything in the `Math` object?

It's time to visit this site.

`https://developer.mozilla.org/en-US/docs/Web/JavaScript`!

This page offers a valuable tool: it is an encyclopaedia of JavaScript objects as well as a reference for the language. Bookmark this site; we will use it a great deal. On this page, expand the References drop-down. Scroll down and click on `Math`. Here you will see a listings called Properties and Methods. All of the properties listed here are just various useful mathematical constants such as e, π, $\sqrt{2}$, and $\ln(10)$. Type them into the console and see them appear. Do the same with your scientific calculator. The word "Method" just means a function that is owned by an object; in this case the methods are just all of your favourite math functions. The reason you use the `Math.` is that they belong to the `Math` object. This is just the possessive (Genitive) case in JavaScript.

If you click on an item in the left-hand column, the right-hand portion of the page will give you a brief lesson on using that method. Let us go over the page for the cosine function in detail. You should look at some of the other pages and experiment with them in the console as well.

Here are the basic sections, blow by blow. The Syntax section shows you the usage of the function. You pass it some value `x`.

Syntax

```
Math.cos(x)
```

The next section tells what sort of value you may pass. In this case, that value must be a number. That is handy, because number is the only type we know about as of now.

Parameter

```
x
    a number given in radians
```

Next, come some examples. Type these into the console and do a little experimentation yourself.

Examples

```
Math.cos(0);           // 1
Math.cos(1);           // 0.5403023058681398

Math.cos(Math.PI);     // -1
Math.cos(2 * Math.PI); // 1
```

Don't worry about the Specifications area yet. The last item shows what browsers understand the function. They all understand cosine. These exercises will cause you to look through this documentation to use these functions. In addition to doing these, take a tour through here. Compare with your calculator and see if there is anything you can do on your calculator you cannot do here.

Programming Exercises

1. Write an expression that finds the largest of 8, 5 and 2.
2. Compute $\log_2(1000)$. Compute $\log_2(1000000)$. What do you notice?
3. Evaluate the expression $e^{2.3} - \cos(1/3.5)$.
4. Evaluate the last expression to the nearest integer.
5. Compute 5.2^3.
6. If a right triangle has legs of length 4 and 5, find the length of the hypotenuse. There is a method in the `Math` object that will do this for you!

2 The JavaScript Type System: Strings

Computing is not just numbers, at least to us humans. We interact with a computer an awful lot using the keyboard; this means that there must be some provision for text and characters to be stored. Take a look at this little console session.

```
typeof("some words")
"string"
typeof("s")
"string"
```

There is an object type called "string;" the purpose of the string type is to store globs of text. Let us explore some of its basic capabilities. Strings can be concatenated (glued together) using `+`. Here is a simple example

```
> "abc" + "def"
<- "abcdef"
```

You can concatenate several strings this way. Try it now.

This is also interesting.

```
> "abc" + 4
<- "abc4"
> 5 + "abc"
<- "5abc"
```

If you use a string and a number as operands, the number gets converted to a string and the results are concatenated. What if you try to multiply a number and a string? Let's see what JavaScript does.

```
> "abc" * 4
<- NaN
```

You get back `NaN`, a symbol meaning "not a number." JavaScript could not make sense of this, so it punted. Ironically, the `NaN` object is of type number! Bear witness to this heinous spectacle.

```
> typeof NaN
<- "number"
```

2.1 Can we get a chunk of a string?

For starters we can see characters in strings by *indexing into* a string. There are two ways to do this.

```
> "abc"[2]
<- "c"
> "abc".charAt(2)
<- "c"
```

At this juncture, you might be saying, "Hey pal, are you trying to pull a fast one? The second character in `"abc"` is a `b`!" The number 2 you are specifying is an `index`. The indices of a string live *between* the characters in the string. Look at this little diagram.

```
-------------
| a | b | c |
0---1---2---3
```

When we specify an index, the character just to the right of that index is fetched. Notice how the indices lurk between the characters. This may seem weird at

first but you will see that, in the end, it makes a great deal of sense. Check this little session out. Henceforth we will omit the annoying <-s in the interactive console sessions.

```
> "abc".substring(1,3)
"bc"
> "abc".substring(1)
"bc"
> "abc".substring(0,2)
"ab"
> "abc".substring(0,2) + "abc".substring(2)
"abc"
```

The `substring` method grabs a substring between the two indices specified. If only one index is specified, the second one is assumed to be the length of the string. Compare these results to the little diagram above and you will say "Ah, makes sense." The concatenation at the end should be a real convincer. Now you know why the indices occur between the characters.

Let us now look at the documentation page for `substr`. It has some new twists that need explanation. We show the beginning of this page here.

The `substr` method returns the characters in a string beginning at the specified location through the specified number of characters.

Syntax str.substr(start[, length])

The fact that `length` appears in brackets means it is optional. The parameters are indices.

Parameters

```
start
    Location at which to begin extracting characters.
    If a negative number is given, it is treated as
    strLength + start where strLength is the length of
    the string (for example, if start is -3 it is treated
    as strLength - 3.)
length
    Optional. The number of characters to extract.
```

The index `start` is a character index. The index of the first character is 0, and the index of the last character is 1 less than the length of the string. substr() begins extracting characters at `start` and collects length characters (unless it reaches the end of the string first, in which case it will return fewer).

If `start` is positive and is greater than or equal to the length of the string, substr() returns an empty string.

If `start` is negative, substr() uses it as a character index from the end of the string. If `start` is negative and `abs(start)` is larger than the length of the string, substr() uses 0 as the `start` index. Note: the described handling of negative values of the start argument is not supported by Microsoft JScript.

If `length` is 0 or negative, substr() returns an empty string. If `length` is omitted, `substr()` extracts characters to the end of the string.

Strings have many methods; you can see them in the documentation. These methods allow you to search a string. Visit this page

`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String`

and take a look. Use what you see to attempt the exercises here.

Programming Exercises

1. How would you find the index of the first letter `a` in the string `"platypus"`?
2. How would you find the index of the last `t` in the word `"spaghetti"`? Find the index of the first `t`, too.
3. How do you make all alphabetical characters upper-case in a string? lower case? Test this on an example. What happens if the string contains non-alphabetical characters. Make one and see.

3 The JavaScript Type System: Boolean

The Boolean type contains two objects, `true` and `false`. There is a unary prefix operator !, which negates booleans. Here is its truth table.

P	!P
T	F
F	T

JavaScript also hasd two binary infix operators, `&&` and `||`. Below we show their truth tables.

P	Q	P	P||Q
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

So if `P` and `Q` represent booleans, then `P&&Q` is true if both `P` and `Q` rare true. And is true if at least one of `P` and `Q` r are true.

4 Variables and Expressions

A variable is a name that can be attached to an object. The rules for these names are simple, so let us begin there.

1. The first character in a variable name can be any alphabetical character or an underscore `(_)`.
2. Subsequent characters can be alphabetical, numerical, or underscores.

Here are some examples of good variable names. Note the camel and underscore styles. Also note that variable names are case-sensitive. The names `abc` and `AbC` are different variables.

```
number_of_clicks    underscore style
numberOfClicks      camel style
x1
slope
employeeID
```

Here are some examples of bad variable names and their faults.

```
semi;colon          illegal punctuation
space cadet         spaces not allowed
2b_or_not2b         illegal to start with a digit
black&white         illegal special character
gone.dotty          illegal period
```

JavaScript, like every prgramming language, has languate keywords. Do not use them for variable names. They change color when typed into a smart editor such as vim, Notepad++, or Atom.

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

All starred entries are new to the latest version of JavaScript (ECMA6). Now let us see how to use variables. To attach a name to a variable, you use the *assignment operator* `=`. Let us show a little sample session. Note the use of the `var` keyword, which says, "I am creating a variable."

```
>>  var x = 4
4
>> var cow = "guernsey"
"guernsey"
>> var y = 14
14
```

Read the `=` as "gets." Writing `x = 4` binds the name `x` to the number object `4`. Now see this.

```
>> x
4
>> y
14
>> cow
"guernsey"
```

Terminology Interlude These terms will become a part of our standard vocabulary. They describe things we will deal with quite often and provide a useful and succinct means for discussing computer languages.

- **token** This is an atom of meaning; it is an irreducible unit of meaning in a language. Object literals such as `5`, `4.2`, or `"cows"` are tokens. Variable names are also tokens. Variables and language keywords (we will see these soon) are tokens, too.
- **expression** An expression is a combination of tokens and operators. For example, the expression `x + 5` contains the tokens `x`, `+`, and `5`.
- **operand** An operand is something being acted upon by an operator. An example of this is `"some" + "thing"`, where the two strings are operands of the concatenation operator `+`.
- **unary operator** This is an operator taking one operand. An example of one of these is `-` in the expression `-x`. This operator changes the sign of a number object.
- **binary operator** This is an operator taking two operands. Many arithmetic operators are binary operators. These include `+`, `-`, `*`, `/`, and `%`.
- **infix** A binary operator is infix if it occurs between its operands. All of the operators just above are infix.
- **prefix** A binary operator is prefix if it occurs before its operands. The change-sign operator `-` is a prefix unary operator, as is the negation operator `!` for booleans.
- **postfix** A binary operator is prefix if it occurs after its operands.
- **parse** To impose meaning (upon symbols) according to stated grammatical rules.

Now we are ready to see expressions at work, continuing from our previous session. In this interactive prompt, we see that when a variable's name is entered, the object bound to that variable gets printed. If you enter an expression, that expression will be evaluated. We show this here.

```
>> x*y
56
>> x+y
18
>> x*Math.pow(y, 2)
784
```

So when you give JavaScript `x*y`, it does the following

1. You have given `x` and `y` the values 4 and 14. These are then recorded in the *symbol table*, which is a dictionary of variables and the values they are bound to.
2. JavaScript sees the expression `x*y`, fetches the values for the two variables from the symbol table, and substitutes the values bound to `x` and `y` to get `4*14`.

3. This expression is parsed and evaluated; it is found to have value `56`.
4. JavaScript hands the result, `56`, back to you.

The symbol table for our session looks like this.

```
x -> 4
y -> 14
cows -> "guernsey"
```

The symbol table contains all information that is visible to you.

Really, There is more than one symbol table in JavaScript that is visible when you are using expressions. A second symbol table that is visible to you is the *global symbol table*; things in this symbol table are always visible.. It is where the `Math` object lives.

You might ask, "What if I try to use something not in a visible symbol table?" Let us try that and find out.

```
> quack
quack
VM919:1 Uncaught ReferenceError: quack is not defined()
```

The console sends you an error message that the variable `quack` is not defined. You should deliberately make mistakes in the console and get used to seeing the error messages they generate. This will help you when you are trying to figure out what is going wrong.

5 Seeing JavaScript in a Web Page

So far we have just placed code in the console and had it evaluate simple expressions. This is, after all, purportedly, a book on web development. So it is logical to ask: *How do I get JavaScript to display on a web page?*'

Begin by making a skeleton HTML page. Place a paragraph element with the `id` of `"showJS"`. You can use any ID you wish.

```
<!DOCTYPE html>
<html>
<head>
<title>See JavaScript Run!</title>
</head>
<body>
<p id = "showJS">Your JavaScript will appear here.</p>
```

```
</body>
</html>
```

Next, add a `script` tag at the bottom.

```
<!DOCTYPE html>
<html>
<head>
<title>See JavaScript Run!</title>
</head>
<body>
<p id = "showJS">Your JavaScript will appear here.</p>
<script type = "text/javascript">
    x = 5;
    y = 4;
    document.getElementById("showJS").innerHTML = 5*4;
</script>
</body>
</html>
```

Now open this page with your browser. Here is what happens. The browser loads this page staring at the top. It digests the `head` element and knows to put a title in the title bar. Then the `body` element opens and the paragraph inside is constructed. Next, the `script` tag says the browser, "Yo this is JavaScript!" Regard the contents of this element as such. So now let us step through this script. We begin by assigning values to the variables `x` and `y`.

When you are coding on a page, as opposed to working in a console, you need to place semicolons at the end of certain statements, known as "worker statements." Worker statement have the property that, if you read them, then form a grammatically complete sentence.

Now we come to the mysterious statement on the next line. The symbol `document` refers to the page the script is on. So think of `document` as meaning "this page."

This symbol `document` is actually a JavaScript object; it is the web pages your JavaScript is running on.

The type of this object is a `node`; a node can be any element on a web page. A node is an object that knows its type (tag type), its attributes, its HTML contents, and its style properties. A node's properties can be altered by JavaScript.

We now invoke the method `getElementById`, which now refers to the paragraph element present in the document. Elements on pages are also JavaScript objects. The `.innerHTML` now points at the HTML inside of the paragrah element. All that is present is a text element.

Finally, the text inside of the paragraph is replaced by 20, the result of evaluating the expression `5*4`.

This seems like a lot of new ideas at once, but we will use this pattern repeatedly as we learn JavaScript so we can display what we are doing. Notice how the use of `id` allows us a simple way to specify a location on a web page.

`More about JavaScript Expressions` We have met the assignment operator `=`, which we used to give values to variables. Let us say a little more about it. Make a new console session. We are going to learn a little more about assignment.

In general, you can assign a variable as follows

```
variableName = expression
```

Whatever expression you place on the right-hand side is evaluated, the then the variable `variableName` is bound to it. Let us now see an example.

```
> x = 5*4
20
> x
20
> x = x + 10
30
> x
30
```

Here in the first line, we assigned `5*4` to the variable `x`. Notice that in an assignment, JavaScript works from right to left. Technically, we say that the assignment operator *associates from right to left.*

For those of you used to seeing `=` used in a math class, the statement

```
x = x + 10
```

has a shocking appearance. But, you must remember that `=` does not mean "equals;" you should read it "gets." What happens here is that `x + 10` is first evaluated. Our symbol table has the entry

```
x -> 20
```

This value is fetched and `x + 10` evaluates to `30`. That value, in turn is now bound to `x`. The old value of `20` is overwritten and it is gone. The symbol table now contains the entry

```
x -> 30
```

Hence, the appearance of the output. So, it is not at all unusual to see a variable on both sides of an assignment.

You might ask, "But what about equality?" This brings us to a new infix binary operator `==`. Enter this in your console session.

```
> 2 + 2 == 4
```

and when you hit enter you get this reply.

```
> 2 + 2 == 4
true
```

The operator `==` is the *isequalto* operator. It is an infix binary operator. If the two operands contain the same value, the isqualto operator will reply `true`; otherwise it replies `false`.

The tokens `true` and `false` are the *boolean objects*. You can see this using the `typeof` operator as follows.

```
> typeof true
"boolean"
```

So now we have a new species `boolean` for true/false values. This brings up a new set of operators that we will find useful, the `relational operators`. This table describes them.

`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to
`!=`	not equal to
`==`	isequalto

These are all binary infix operators. They all reply with a boolean value. If the operands are numbers, the comparison is the standard numerical comparison.

A Weirdism JavaScript has an additional operator, `===`, which we demonstrate now.

```
> 5 == "5"
true
> 5 === "5"
false
> "true" == true
```

```
false
> 5 !== "5"
true
```

Note the existence of the `!==`, which is the negation of `===`.

The `===` operator tests for "same value and same type" for numbers. What is interesting here is that `true == "true"` evaluates to `false`.

The suthor speculates that since a lot of information from the web comes in from forms in the form of strings, the fact that `5 == "5"` evaluates to `true` is a handy convenience.

Also, take note of this quirk.

```
> true == 1
true
> true === 1
false
```

What about strings and relational operators?? Let's start simple.

```
> "a" < "b"
true
```

That looks like what we expect. Now let us try this.

```
"Z" < "a"
true
```

Uh oh. What happened?

Let us think about alphabetization of words first. How do you determine which of two words is first in the alphabet? You do something like this. Let's denote the words by `x` and `y`.

1. Look at the first character of the two words. If the first character of `x` comes before that of `y`, we know `x` comes first alphabetically.
2. If the first character of `x` comes later that of `y`, we know `x` comes second alphabetically. For example, `"artichoke"` comes before `"beet"` because `"a"` comes before `"b"` in the alphabet.
3. If the first characters are the same, continue checking subsequent characters until you find a different character. Then do the first two steps on that character. An example of this is that `"lambaste"` comes before `"lambchop"`.

4. If one word runs out before the other and before a conclusion is made, then the word ending first comes first alphabetically. For example `"cow"` comes before `"cowabunga"`.

This kind of ordering is called a `dictionary` or `lexicographical` order. What JavaScript does with the characters in a string is called *asciicogaphical ordering*. It is a lexicographical order, but characters are compared by their ASCII (byte) values. Since the character `"Z"` has ASCII value 90 and `"a"` has ASCII value 97, you can see how the result arises.

6 Please Excuse My Dear Aunt Sally

JavaScript has many operators and they have an order of operations. The arithmetic operators `+`, `-`, `*`, and `/`, have the usual order of operations. They associate from left to right, just as the do in algebra. Mrs. Wormwood would say, "You first deal with parentheses, then exponents, then multiplication from left to right, then addition and subtraction from left to right. The highest precedence, as in algebra, is held by parentheses. When in doubt, use them.

The relational operators have lower precedence than the arithmetic operators, so an expression such as `4 + 5*8 > 3*7` crunches down as follows. Below you can see each step that occurs; after the expression, the operation carried out on that line is indicated.

```
4 + 5*8 > 3*7   start
4 + 40  > 21    multiply
44      > 21    add
true            compare
```

Assignment has an even lower priority. So if you do this `x = 5*7 > 6*6` it crunches like so.

```
x = 5*7 > 6*6   start
x = 35 > 36     multiply
x = false       compare
                That's all folks!
```

As a result, the variable `x` is bound to the boolean value `false`. So far, we have this hierarchy. The highest lines in the table have the highest precedence.

Operator(s)	direction
`()`	left to right
`.`, as in `"foo".charAt(1)`	left to right
`* / %`	left to right
`+ -`	left to right
`=`	right to left

Programming Exercises

1. Determine which is bigger, π^e or e^π.
2. Tell what items are in the symbol table after this code runs

```
var x = 15;
var y = 17;
x = x*y;
var oahu = "lei";
var hilo = oahu + " of orchids";
```

3. Place these items in asciicographical order. If you have a one-character string, say `x`, you can determine its ASCII code using `x.charCodeAt()`.

```
Zebra
aardvark
cat
Cat
17 32
```

You can check your answer by entering these into the console.

```
> var a = ["cat", "Cat", "Zebra", "aardvark", "17", "32"];
> a.sort()
> console.log(a)
```

7 More about Objects

Objects have three important attribues: state, identiy, and behavior. We will discuss the objects we have seen so far and learn about these attributes for them. Here, succinctly, is the deal.

1. identity: This is what an object IS. Objects are just chunks of memory. They contain data for their state and code for their behaviors.
2. state: This is what an object KNOWS.
3. behavior: This is what an object DOES.

Let us look at all of the types of objects we have seen so far and analyze their state, idendity, and behavior. Identiy is the same for all objects: they are regions of storage in a computer's memory.

A number object's state is simply the value it is storing. Numbers behave as expected in the presence of arithmetic operators such as +, -, *, /, and %.

A string object's state is the chararacter sequence it contains. Strings have many behaviors. You can extract characters as one-character strings using `charAt()` and `[]`. You can get a piece of a string using `slice` or `substr`. You can get a copy of a string in lower or upper case. The JavaScript String reference page has a complete list.

A boolean's state is just `true` or `false`. Booleans do their thing in the presence of the three boolean operators, not (`!`), and (`&&`) and or (`||`); this is their behavior.

Functions are more complex. To describe a function completely, we need to describe a function's preconditions and postconditions. This describes a function's action and required inputs.

A function's preconditions indicate what should be true before a function is called. This includes such things as the number and types of inputs (arguments) it takes.

A function's postconditions indicate what is true if when a function is done executing. It can have a *return value*, or output, and it can have *side effects*, things that are done and which persist beyond the function's execution. We will talk about this in greater depth in the next chapter.

So far, all of the objects we have learned about are *immutible*; this means that their state cannot be changed after they are created. When we re-assign a variable, we just point at a new object, possibly orphaning the old one.

Nodes are JavaScript objects. The node is the first example of a mutable object; JavaScript can change the state of a node. A node's state includes the following.

- the tag type of the element
- the attributes of the element, including its *id*.
- the style properties of the object

Nodes can be altered by various JavaScript methods such as `document.getElementById`, which can change their state.

Type	**State**	**Behavior**
number	value	Numbers behave as expected in the presence of arithmetic operators
string	character sequence	Strings exhibit a rich panoply of behaviors such as `charAt`, `[]`, and `slice`.
boolean	`true`/`false`	Booleans behave as expected in the presence of not (`!`), and (`&&`) and or (`||`).
function	its preconditions and postconditions	its action
node	tag type, attributes, style properties	A node's state can be changed by various means.