hide Table of Contents

Overview

This reading is not a tutorial on programming. Instead, it's a quick introduction to JavaScript intended for someone who understands at least one programming language (preferably with a syntax like C or Java, since JavaScript shares their syntax) and the associated concepts of variables, functions, conditionals, loops and data structures like arrays and hash tables. In addition, JavaScript supports Object-Oriented Programming, so familiarity with the concepts of objects, classes, and methods is helpful, though this reading doesn't cover implementing your own objects, classes and methods.

About JavaScript

JavaScript is a dynamic language that is widely supported by modern web browsers. It is similar syntactically to Java, but semantically resembles Lisp and Python, in that it is weakly typed, and has anonymous functions and closures, among other differences. Don't be confused by the fact that its name starts with "java": it has no connection to the Java language other than those four letters.

Learning JavaScript

If you'd like a more thorough introduction to JavaScript, you could consult some of these references. This StackOverflow page lists some good resources to learn JavaScript. I also have some books I can lend. My opinions:

- JavaScript Tutorial. Kelsey recommends this tutorial.
- The Douglas Crockford book, *JavaScript: The Good Parts* is delightful, but is extremely concise and gets into some deep theoretical issues. It's not an easy read.
- I haven't viewed the Crockford videos, but I would expect them to be very good.
- I would bet that the *Eloquent JavaScript* online book would be the way to go if you decide that Crockford isn't for you.
- CS110 uses David Sawyer MacFarland's *JavaScript and jQuery: The Missing Manual*, 2nd edition, which I chose because I liked it a lot. You won't need the jQuery part very much.
- Here's a JavaScript Quick Guide. Just 13 intense pages, in a printer-friendly format.

In the following sections, I'll review a few essentials which will help those of you who just need some reminding, but this is *not* complete, and you should definitely avail yourself of some of the resources above if you find this too confusing.

The SCRIPT tag

You switch from HTML to JavaScript (JS) using the **`script`** tag. If you use the **`src`** attribute, you can load JS code, including the jQuery library, from another file or URL, but then the body of the **`script`** element is ignored.

Basic Syntax

JS uses syntax like Java, so both **`// comment`** and **`/* comment */`** work as comments. Statements look like Java, ending in semi-colons. (JS has a mis-feature that allows semi-colons to be optional by allowing the parser to guess where they should be; don't use this mis-feature, because an incorrect guess is hard to debug.)

```
/* omit this dumb code
  var x = 3;  // magic number is 3
  var y = 7;  // another mystical number
*/
```

Major control structures (`if` statements and `for` and `while` loops) look just like Java. The following nested loops with conditionals prints the prime numbers from 2 to `max`.

```
var max = 50;
var i,j;
for( i = 2 ; i < max ; i++ ) {
    // loop over possible factors up to sqrt of i
    j = 2;
    var prime = true;
    while( prime && j*j <= i ) {
       if( (i % j) == 0 ) {
           // console.log(j+" evenly divides "+i);
           prime = false;
       }
       j ++;
   }
   if( prime ) {
       console.log("prime: "+i);
   }
}
```

execute it

Useful tools for debugging are built-in functions `alert` (which halts the program and pops up a window that must be acknowledged) and `console.log` which writes its argument to a hidden window that you can find if you poke around in your browser a bit. Google for “browser javascript console” for your browser.

As of this writing:

- In Google Chrome on a Mac, Command-option-j opens the JavaScript console.
- In Firefox on a Mac, Command-option-k opens the Web Developer Console
- In Safari on a Mac, Command-option-c opens the Error Console.

Go ahead and open up your JavaScript console using one of those commands. You can then copy/paste the following into your console to see what it does, or click the button below it. (Clicking the button executes the code in a slightly different environment, so the x and y variables won't be available in your console, but the console.log function works correctly.)

```
var x = 3;
var y = 4;
var sum = x + y;
console.log("the values are "+x+" and "+y);
alert("the sum is "+sum);
console.log("they add up to "+sum);
```

execute it

Notice the difference between `alert()` and `console.log()`. Note also that `alert()` stops the browser, so the second `console.log` doesn't happen until you click in the alert window.

Datatypes

JavaScript has a few scalar datatypes:

- strings (delimited by either single or double quotes)
- numbers, notated in the usual way
- booleans, notated by the bare words `true` and `false`.

Like Java, the string concatenation operator is **+** (the plus sign). Unlike Java, the **+** operator can be either *add* (if both operands are numbers) or string concatenation (if either operand is a string), and this isn't known until run-time, since JavaScript is dynamically typed. This is an ugly facet of the language, causing innumerable mistakes, so be careful about the type of data stored in your variables. If you're not *sure* whether the variables will contain numbers or strings, you will have no way to know what the result will be.

This illustrates an easy mistake to make:

```
var x = 3;
var y = 4;
alert("the sum is "+x+y);
```

execute it

Compound Datatypes

JavaScript has two beautiful compound datatypes, arrays (also called *lists*) and objects. Both have a simple and convenient literal syntax:

```
// an array of some primes
var primes = [2, 3, 5, 7, 11, 13 ];
console.log("we currently have "+primes.length+ " primes.");
primes[6] = 17;  // add another
primes.push(19);  // and another
console.log(primes);
```

execute it

(Remember that if you want to play around with the **`primes`** variable, you should copy/paste that code into the JavaScript console, rather than using the button.)

As you would expect, arrays are zero-based and indexed by integers. They support a number of useful array methods as well.

In JavaScript, an “object” is a data structure of name-value pairs, what other languages call hashtables (Lisp, Java), associative arrays (PHP), dictionaries (Smalltalk, Python), hashes (Perl), and so forth. (In a separate reading, we'll learn why they are called objects.) Here's an example:

```
// an object representing a movie
var movie1 = { title: "Dr. Zhivago",
               director: "David Lean",
               starring: ["Omar Sharif", "Julie Christie", "Geraldine Chaplin"],
               release: 1965 }
movie1.running_time = 197; // in minutes
console.log("The title is "+movie1.title);
```

execute it

Note that values of the properties can be scalar or compound; above we have an array literal of strings for the stars of the movie.

This object literal syntax is so compact and useful that it is now one of the more common representations of data on the web, called JSON or JavaScript Object Notation.

You can convert a JavaScript data structure of strings, numbers, booleans, arrays and objects into a string (suitable for printing or sending/receiving over the web) using the function **`JSON.stringify`**. You can convert such a string back into the data structure using the function **`JSON.parse`**.

```
var movie1 = { title: "Dr. Zhivago",
               director: "David Lean"}
var movie_string = JSON.stringify(movie1);
var movie_copy = JSON.parse(movie_string);
if( movie1 != movie_copy ) {
    console.log("The copy is a different object from the original");
}
if( movie1.title == movie_copy.title && movie1.director == movie_copy.director ) {
    console.log("But they have the same title and director");
}
```

execute it

An important note on terminology: this concept of turning a data structure into a string, suitable for writing to a file or transmitting across a network, and then reversing the operation to re-create the data structure is a common one in computer science, not reserved just for JavaScript. One standard term for it is *Serialization*.

Variables and Scope

In JS, variables don't have types; data does. So a variable can store any type of data, and you don't have to declare the datatype. The following is fine:

```
var x = "five"; // a string
x = 5;  // a number
x = true; // a boolean
x = [2,3]; // an array
x = {a: 2, b: 3};  // an object
```

In *practice*, the code above is awful, because you or anyone reading your code will not have any idea what kind of data is stored in **`x`** at any time. For another example, what kind of data is stored in the following variable?

```
var students;
```

Given an ambiguous name like that, any of the following might be reasonable:

```
students = 3;   // maybe numStudents is better?
students = ["alice", "bob", "charlie" ];  // maybe studentList?
students = "alice, bob, and charlie";
```

Since the datatype isn't there to help clarify, you'll have to be clearer in the naming of your variables and their documentation.

When creating a variable, you can declare it using **`var`**. That is optional, but you should do it anyhow. Do *not* use **`var`** if you are re-assigning to an existing variable.

Functions

Functions in JS have a very simple syntax, since there's no need to declare types of arguments or return values. The following creates a function named **add5** and invokes it on **2**:

```
function add5(x) {
    return x+5;
}

console.log("the result is "+add5(2));
```

execute it

As you can infer, the syntax is:

```
  function nameOfFunction(arg1, arg2, arg3, arg4) {
      // body
      return ans;   // optional
  }
```

JavaScript does not check that a function is invoked with the right number of arguments: you can pass in too many or too few. This allows for some fancy features, like optional arguments, but is also an easy way to make a mistake. Fair warning.

A function does not have to return a value. If it wants to, it uses the **`return`** keyword.

Local Variables

If a function needs a local variable, you can, of course, create them in JavaScript, but you have to be careful to use the **`var`** keyword (which is, unfortunately, optional, so there's no error message if you make a mistake).

A variable is essentially either **`global`** (declared and used outside any function and shared by all of them) or **`local`** (declared inside a function and available only to that function). The distinction is the critical use of the **`var`** keyword.

```
var x = 5;  // global named x

function addToGlobalx(y) {
    x++;         // increments global x
    return y+x;  // refers to global x
}

function addToLocalx(y) {
    var x = 2;   // new local named x
    x++;         // increments local x
    return y+x;  // refers to local X
}
```

```

// global x keeps increasing
console.log(addToGlobalx(3));  // 9
console.log(addToGlobalx(3));  // 10
// but local x is always 2
console.log(addToLocalx(3));   // 6
console.log(addToLocalx(3));   // 6
```

execute it

(There are other scopes, such as closures, but we won't make much use of them for now.)

Anonymous Functions

You can also have an *anonymous* function. The following creates an anonymous function that adds four to its argument. The anonymous function is stored in a variable called **add4**. The last line invokes the function on **3**:

```
var add4 = function (x) { return x+4; };

console.log("the result is "+add4(3));
```

execute it

Functions and variables share the same *namespace*, so the preceding way to create a function is (nearly) the same as the first way we saw, namely:

```
function add5(x) {
    return x+5;
}
```

(The only differences are in minor ways such as error messages, where an error message from a named function can say what the name of the function was.)

In the examples above, is **add5** a function or a variable? It's both: **add5** is variable whose value is a function named **add5**. What about **add4**? Is **add4** a function or a variable? It's both, because it's a variable that contains an anonymous function, but since the variable name can be used anywhere that a function name is used, including in invoking the function, **add4** is a function.

Anonymous functions are used *a lot* by jQuery programmers, so it's a good idea to get comfortable with seeing them if you're in CS 304. We will probably not need anonymous functions in CS 307, though we may want to use local functions (see below).

Local Functions

Local variables and functions as values give us a nice way to have *local functions*. Local functions allow us to define a function that is useful as a helper to a global function, but which you may not want to define globally. Here are some simple examples:

```
function isPythagoreanTriple(a,b,c) {
    var square = function (x) { return x*x; };

    return square(a)+square(b) == square(c);
}
console.log(isPythagoreanTriple(3,4,5)); // true
```

execute it

A common use of helper functions is in recursion, particularly tail-recursion. Here's a classic example:

```
// tail recursively compute factorial
function factorialTail(n) {
    function helper(n,result) {
        if (n==0) {
            return result;
        } else {
            return helper(n-1, result*n);
        }
    };
    return helper(n,1);
}
```

```

console.log( factorialTail(4) );  // 24
```

execute it

Functions as First Class Objects

JavaScript, like many civilized languages but not all, has functions as “first class objects”. That means that:

- A function can be stored in a variable. (We saw that with **add4**.)
- A function can be passed as an argument to a function or method.
- A stored function can later be invoked, when desired.

As a silly example, consider the following, which defines a function that returns a number, another function that takes a number-returning function as an argument and returns one more than that:

```
var five = function () { return 5; };
var num = five;
var next = function (curr) { return 1+curr(); };
var ans = next(num);
alert("the answer is "+ans);
```

execute it

Notice that when we pass **five** as an argument, we want to pass the function itself, not the result of invoking it, so we just give the variable that contains the function, without the **()** after it. The parens, which you see when the **next** function invokes **curr**, is what invokes a function stored someplace.

To illustrate this more realistically, consider the following, where we define a function, **reportTarget**, that tells you the text of the target of an event. Next, we use jQuery to attach that function to every **h2** element on this page. The **reportTarget** function is not invoked until you click on an **h2**. (Since that happens within jQuery, we don't see the invocation code, but it's just like the previous example.) Try it!

```

function reportTarget(event) {
   var text = event.target.innerHTML;
   alert("You clicked on "+text);
}

$("h2").click(reportTarget);
```

Functions as first-class objects is used a lot in applications like this, where the function is an *event handler*. Of course, most jQuery programmers would use an anonymous function, like this, which conveniently puts all the code in one place.

```
$("h2").click(function (event) { alert("You clicked on "+event.target.innerHTML);});
```

On the downside, though, look at the blizzard of punctuation at the end of the line!

Functions with Keyword Arguments

If a function takes a great many arguments, it can be inconvenient to use, and the invocation can be hard to read. Unless you've memorized the order of the arguments in the following realistic example, you're likely to find the following a bit confusing:

```
glFrustum(-2,2,1,-1,1,10);
```

(**glFrustum** is related to setting up a synthetic camera in computer graphics.)

Some languages, such as Python, address this by having *keyword* arguments:

```
glFrustum(left = -2, right = 2, top = 1, bottom = -1, near = 1, far = 10)
```

This is a bit more typing for the caller of the function, but the meaning of each argument is clear. Also, the arguments can be done in any order.

If there are good default values for many of these arguments, the function call becomes even easier, because you can just give the ones you need and let the others default.

```
glFrustum(far = 10)
```

JavaScript doesn't have special support for keyword arguments in the language, but the object literal syntax is so easy that programmers use it for keyword arguments.

```
  /* returns the volume of a box with the given dimensions, 'width,'
  'height,' and 'depth.' */

function boxVolume(dims) {
    return dims.width * dims.height * dims.depth;
}

console.log(boxVolume({width: 2, height: 10, depth: 3}));
```

execute it

Note the braces around the object literal, and the whole literal is the **`dims`** parameter of the function.

If you want to have default values, you can do that in the code very easily, using the **`||`** (logical OR) operator, which will use the second value if the first is false, and an undefined property counts as false:

```
  /* returns the volume of a box with the given dimensions, 'width,'
  'height,' and 'depth.' Dimensions default to 1. */

function boxVolumeDefaults(dims) {
    var w = dims.width || 1;
    var h = dims.height || 1;
    var d = dims.depth || 1;
    return w * h * d;
}

console.log(boxVolumeDefaults({height: 10, depth: 3}));
```

execute it

You'll note that I was extremely terse with the variables I used in the previous example. This is justified by the fact that (1) the meaning of each is completely clear from their initialization, and (2) they are never used far from their initialization, so there is no advantage to a longer, more descriptive name.

The Date Object

JavaScript has a reasonably nice and convenient **`Date`** object. The **`Date`** function returns an object that captures the current date and time (from the system clock) and has methods to return those values. (Note that the month numbering is zero-based, so we add one here.)

```
var d = new Date();
var mon = d.getMonth()+1;
console.log("The date is "+mon+"/"+d.getDate());
console.log("The time is "+d.getHours()+":"+d.getMinutes());
```

execute it

Using Objects

The previous section raises the question of how to use objects in JavaScript. Like pretty much all modern languages, JavaScript supports Object-Oriented Programming (OOP), but with some significant differences. In a nutshell:

- It doesn't really have *classes* the way that Java and C++ do
- However, the functionality of classes can be implemented in the JavaScript language and is often done so by packages like Three.js.

I will save for a later time how you can implement your own classes, objects and methods in JavaScript , but feel free to read ahead if you'd like. For now, we'll only look at how the OOP features of JavaScript can be used.

The NEW Operator

One of the fundamental things you do in OOP is to create instances of a class. In JS, as in Java and C++, this is done with the **new** operator/keyword. We saw this above with the Date object. Here's another example:

```
var d1 = new Date();
alert('How fast can you click "ok"?');
var d2 = new Date();
var diff = d2.getTime() - d1.getTime();
console.log('It took you '+diff+' milliseconds to do so');
```

execute it

Note that as a *convention*, any function with an initial capital letter is a constructor whose invocation is intended to be preceded by the **new** keyword. We see that with the Date objects above.

In CS307, we will often create instances of Three.js stuff, like this:

```
  var box = new THREE.BoxGeometry(w,h,d);
```

Invoking Methods

As we've also seen, to invoke a method on an object, you do this:

```
objvar.method(arg1,arg2,...);
```

That is, you give a variable containing the object (like **objvar**), a dot, the name of the method, and then any arguments, in parens.

More generally, you can replace the variable with any expression returning an object, though the resulting code can sometimes be hard to read. For example, using the **getTime** method on dates:

```
console.log('Since the epoch, it has been '+(new Date()).getTime()+" milliseconds.");
```

execute it

Or, using the **substring()** and **toUpperCase()** methods on strings.

```
var list1 = ["fat", "fit"];
console.log("rhymes are "
  +(list1[0].substring(1).toUpperCase())+" and "
  +(list1[1].substring(1).toUpperCase()));
```

execute it

In CS307, we can invoke methods on Three.js stuff like this:

```
  var box = new THREE.BoxGeometry(w,h,d);
  box.translateX(dist); // move it to the right
```

How can you find out what methods an object supports? Beyond reading the documentation, you can also use the JavaScript debugger. Given a variable containing an object, typing the variable name and a dot into the debugger will typically give you a menu of methods and properties.

Try it! Find out the methods on strings, dates and lists using the following example, copy/pasting them into your JavaScript console and then typing the variable name and a dot:

```
var a_string = "JavaScript is fun";
var a_list = ["and", "also", "useful"];
var a_date = new Date();
```