Lecture 1: Introduction to JavaScript

Nick Hynes

January 9, 2014

Topics

1 Introduction

Welcome to the first lecture of Introduction to JavaScript! Let me begin by introducing myself: my name is Nick Hynes and it is my great pleasure to be your instructor for this course. Joining me is your wonderful TA, Cassandra Xia. She will be available to help you with your JavaScript-related queries during the peer-debugging sessions.

During this hour of the class, you'll be participating in an interactive lecture powered by Live Lecture (http://introjsiap.com/livelecture). You can use this to ask clarifying questions or let me know that you want something explained in more detail. This is experimental, though, so your feedback will determine its further use.

During the second hour, you'll be working in pairs or small groups to debug each other's code or work on the current assignment. If you have any questions, you can ask me or Cassandra. The goal is to allow you to practice material learned in lecture while hopefully gaining insight from different implementations. This, too, is experimental. Your feedback on issues

encountered during debugging will used for research Tom Lieber and I are doing on ways to make JavaScript debugging more user-friendly.

Well, that's all I had as far as introductions go, so without further ado, let's begin!

2 What is JavaScript?

JavaScript is an object-oriented, weakly-typed, interpreted language that is used to write client and server applications. Now that you're convinced of my ability to regurgitate technical jargon, let's go over what they mean, and, more importantly, why they matter to you.

JavaScript is interpreted. This means that the code you write is translated into commands that are executed by the JavaScript "virtual machine" – most notably, your browser. The "virtual machine," or engine, provides a set of JavaScript objects that you can manipulate to change the state of the external environment, namely the HTML DOM on the client or the filesystems and networks of the server. It's almost like the levers used to operate a bulldozer: you move the controls and the engine takes care of the rest; the main difference is that the objects you're bulldozing are JavaScript, not rocks. Of course, the controls provided by the VM (particularly on the client) are designed to keep you from wreaking too much havoc on the user or the computer. This may have been a bit abstract, so here's a visual. *See lecture slides.*

As for weakly-typed, practically this means that, like Python and unlike Java, any type can be assigned to any variable. This is both good and bad, as, although you need to keep track of the types of variables in your system, you get to focus on their capabilities.

3 Data Types

3.1 Objects

If you've ever gone through any other language tutorial, starting the "Data Types" section with objects might strike you as odd. However, in JavaScript *everything* is an object. "Really? Everything?!" Yes *everything*! Though some may present different interfaces, every data type has an object representation.

What, then, is an object? An object is, at its very core, a key-value store. You're already familiar with this if you've ever worked with a hash table or Python dictionary. The keys, or *properties*, are Strings and their values are objects of any type, since JS is weakly typed. Because of this, the "type" of an object is based entirely on its properties. It helps to think of objects in terms of what they can do rather than what they are.

The language also defines some classes of objects that have default functionality. These constitute the primitive type system of the language. Let's look at them now.

3.2 Numbers

In general, `Numbers` are integers or double-precision floating point numbers – that is, numbers containing a decimal point.

```
42                 //A Number (an integer)
6.28318530717959   //Also a Number (a double)
1e-7               //A double in scientific notation

2i                 //Not a Number
notnumber          //Not a Number
```

Operations on Numbers work in much the same way as most other languages:

```
1 + 1 = 2

1.6666666667 - .3333333333 = 1.3333333335 //Rounding error!

21 * 2 = 42

-1024 / 32 = -32

5 % 3 = 2 //%, the modulus (remainder) operator
```

There are also two special cases that you want to avoid, or for which you want to check. Instead a raised error or thrown exception, performing an invalid mathematical operation results in one of these:

```
Infinity          //The result of dividing by zero
NaN               //The result of something crazy like √-1
```

The best part is that, since you can perform operations on these weird numbers, any bugs they cause will likely be far removed from the original source of the problem! Here are some examples, for your amusement.

```
Infinity (-,*,/) real number = Infinity
Infinity % any number = NaN
Infinity + Infinity = Infinity
Infinity - Infinity = NaN
Infinity * Infinity = Infinity
Infinity / Infinity = NaN

NaN (-,*,/,%) any number = NaN

Any number (-,*,/,%) a number = NaN //+ is special
```

3.3 Strings

Strings in JavaScript are similar to those found in Java and Python (and most languages). They're, quite literally, strings of characters. JS strings are immutable, so each operation produces a new string, leaving the original String unchanged. Also, every object has a String representation. Incidentally the String representation of a String is (a copy of) itself.

3.4 Booleans

Without booleans, it is likely true to say that the statement "conditionals make sense" would be false. A boolean is a value that is either `true` or `false`. You generally use these in conditionals. In JavaScript, while the Booleans, themselves, make sense, their conversions from other types are not as clear.

Data type	Value	Boolean value
Number	0	`false`
	not 0	`true`
	NaN	`false`
	Infinity	`true`
String	empty string	`false`
	non-empty string	`true`
Object	any other object	`true`
None	undefined	`false`

Fortunately, you'll create most of your Booleans either by explicitly setting them to `true` or `false`, or by using the comparison operators.

3.5 Arrays

Arrays are most similar to Python lists, but an object-based explanation will likely be more meaningful. Essentially, an Array is an object that provides methods that operate on the non-negative integer-valued properties, or indices. This also means that, as with ordinary objects, you can assign any object to any index in an Array whether or not the previous indices have been filled. This also means that you can do some interesting (and possibly hacky) things with Arrays. You'll see some examples of this in the section on syntax.

3.6 Functions

In the spirit of saving the best things for last, I present to you, functions! Basically, a function is a special object that is "callable," meaning that it's an executable block of instructions. Like in many other languages, functions can take arguments and produce a return value. Since functions are also objects, they can be assigned to a property of an object and be passed into and returned from other functions.

Function objects also include a *prototype* Object that can be used to assign properties to all other Objects that were constructed using the Function.

4 Syntax

Now that you're familiar with the type ecosystem, let's look at how you can interact with it. Of course, no introduction to any programming language would be complete without a "hello world" of some sort. Let's see how much utility we can eke out of this programming paradigm.

Example 1 – A simple statement

```
1 /* Displays an alert box (read: obnoxious modal dialog)
2    containing the text "Hello, world!" */
3 alert("Hello, world!"); //Make the magic happen
```

Okay, not bad. On lines 1 and 2, there's an example of a multi-line comment and there's a single-line comment on line 3. Generally, these look like /*...*/ and //..., respectively. The tastiest bit, the `alert`, is a function that takes one argument of any type – a string, in this case – and shows its string representation in a dialog box. Although there will be more on functions shortly, it's now important to note that whitespace is ignored and that every statement ends with a semicolon.

Great! Now that the tribute to the programming gods has been made, we can move on to more interesting examples.

4.1 Variables and assignment

A JavaScript variable is a name used to *refer* to an object. As such, you can assign any object to any name and vice versa. The syntax to use when creating variables is

```
var variableName = Object
```

Reading the value of a variable or property that does not exist will result in the special value of `undefined`, which is like `null` or `Nil` in other languages.

Example 2 – Veritable variables

```
var aNumber = 42;          //camelCase
var aWord = "beautiful";
```

Scoping, or determining the part of the code in which a variable exists, is relatively simple in JavaScript since there are only two scopes: local and global. Local variables are defined anywhere in a function and are accessible at any point after their definition in the same function. If a variable is defined outside of a function or without the `var` keyword, the variable becomes "global," and is available to any other script running in the page.

Example 3 – Visible variables

```
var var1 = "I'm a global variable!";

function assumeThisIsWhatFunctionsLookLike () {
    var local = "I'm a local variable.";
    var2 = "I'm a global variable, too!";

    console.log(local); //Outputs "I'm a local variable."
}

console.log(var1);  //"I'm a global variable!"
console.log(var2);  //"I'm a global variable, too!"
console.log(local); //undefined
```

4.2 Objects

Aside from creating one of the special classes of Object, the preferred method of creating an Object is the shorthand notation {}. Again, the keys of an object are Strings and the values are Objects. Key-value pairs are specified as `key:value` and are separated by commas.

Since objects are just collections of keys and values, the only really important operations are adding and accessing keys and values. To access values of

properties, you can use either the *dot* operator or *square brackets* operator. Whereas the dot operator can only access properties of un-spaced Strings that don't start with numbers, any string can be placed within brackets. The resulting reference to a value can be used in the same way as a variable.

Example 4 – The key to accessing an Object's values

```
var anObject = {
    aKey: "a value",
}

console.log(anObject.aKey); //"a value"

anObject["anotherKey"] = "another value";
console.log(anObject.anotherKey); //"another value"
```

Numbers

You've already seen most of the interesting things you can do with numbers earlier, so I'll start describing the `Math` object. The static `Math` object is provided to you by the browser and contains a variety of useful, basic mathematical functions like sin(), round(), and random() and constants like π and e.

Example 5 – Mathematical!

```
var angle = Math.PI/4.30355158;

console.log(Math.sin(angle)); //0.6668696350365613
```

Strings

As a quick recap, new Strings are created using a pair of single or double quotes and any object can be "made into" a String using its toString() function. Strings can be concatenated, or joined, by using the + operator. Keep

in mind that the + operator also adds Numbers, so watch out for bugs when concatenating Numbers with Strings.

Example 6 – Letting the concat out of the bag

```
var pieStr = "I have pie";
var happyStr = 'everything is great';

console.log(pieStr + "; " + happyStr);
//"I have pie; everything is great"
```

Example 7 – JavaScript from the future

```
console.log(2000 + 14 + " is this year");
//"2014 is this year"

console.log("The year is " + 2000 + 14);
//"The year is 200014"
```

Notice how the operands are evalulated from left to right. Once a String is encountered, all +'s turn into concats.

Arrays

Arrays are usually created using []. Since Arrays are just objects with fancy functions, they are accessed in the same way; since the keys are numerical, this requires square brackets.

One important thing to note is that the `length` of an array is equal to its highest assigned index plus one. This is stored as the `length` property.

Example 8 – Arrays make sense!

```
var boringArray = [1, 2, 3, 4];

console.log(boringArray[0]); //1 (zero-indexed)
console.log(boringArray.length); //4
```

4.3 Functions

```
function functionName(argument1, argument2, argumentN) {
    /*code*/
    return value; //This is optional
}
```

The name of a function is functionally (no pun intended) equivalent to a variable. You can pass the name around to other functions or assign it as a property of an object. The function is *called*, however, using parentheses:

Example 9 – First-class functions

```
function stringReturner() {
    return "string";
}

function stringPrinter(stringGenerator) {
    console.log(stringGenerator());
}

stringPrinter(stringReturner); //"string"
```

Not only can you simply call Functions, you can also use them to initialize an object of a given "class" by using the `new` keyword before the function call. This creates a new object and calls the Function with `this` set to the new object. The Function is essentially becomes a constructor. If you're not

familiar with `this`, it's a keyword that means you're referring to the current object.

Objects constructed using a Function also have access to any functions and properties added to its `prototype` object. The syntax for modifying a Function's prototype is:

```
Function.prototype.property = value;
```

Keep in mind that only Functions created using a function declaration or the Function() constructor have the `prototype` object. That is, function expressions can not be prototyped!

Example 10 – 300: JavaScript edition

```
/**
 * Represents a Movie about a topic
 *
 * @param about the topic of the movie
 */
function Movie(about) { //Function declaration
    this.about = about;
}

//Creates a new Object, {}, and calls Movie() with it as this
//then assigns the new, initialized object to boxOfficeHit
var boxOfficeHit = new Movie("JavaScript");

/**
 * Plays a movie about the topic being madness
 */
Movie.prototype.play = function() { //Function expression
    console.log("Madness? This is " + this.about + "!");
}

//Notice that the new object now has the play property
boxOfficeHit.play(); //"Madness? This is JavaScript!"
```

On line 1, there's a function named Movie that takes one argument and sets it as the value of object's `about` property. We make a new `Movie` on line 10.

On line 15, we use the `prototype` object of the Movie function to add a new property to all `Movies`. We then are able to call the function assigned to `play` from `boxOfficeHit`, on line 20, since prototype affects "classes" of, rather than particular, objects.

4.4 Flow control

If you understand flow control syntax, then you're well on your way to learning JavaScript. Otherwise, you're gonna have a bad time!

Conditionals

A basic conditional in JavaScript is your standard if/else/else if statement. Before sallying forth to conquer the armies of unknown values, however, you should take these operators. It's dangerous to go alone.

Name	**Operator**	**Description**
Equals	a == b	casts to common type and compares
Strict equals	a === b	compares value and type
Not equals	a != b	casts and compares
Strict not equals	a !== b	compares value and type
Greater than	a > b	returns true iff a > b
Greater than or equal	a >= b	same as above but including a == b
Less than	a < b	returns true iff a < b
Less than or equal	a <= b	same as above but including a == b

Comparison operators

Name	Operator	Description
And	a && b	returns true if a and b are both true
Or	a || b	returns true if a or b is true
Not	!a	returns true if a is false

Logical operators

Example 11 – Conditional number guessing

```
var randomNumber = "4"; //chosen by fair dice roll
                        //guaranteed to be random
if(a === 4) {
    var response = "The number is 4!";
} else if(a == 4) {
    var response = 'The "number" is a "4"!';
} else {
    var response = "I give up! What was the number?";
}

console.log(response); //"The "number" is a "4"!"
```

Now that you know what the basic conditional looks like, let's look at some of the more exotic, labor saving varieties, namely the `switch` statement and the *ternary* operator. The former is a compact way of expressing long if-else if-else clauses and the latter is useful as an inline (quite literally, in-line) statement. .

Example 12 – Switching it up using `switch`

```
var requestedPage = window.location.hash;
//the part of the URL including and after the #

switch(requestedPage) { //This can be any expression
    case "#home": //Like Python, but whitespace not required
        showHomePage();
        break;
    case "#about":
        showAboutPage();
        break; //Don't forget to include this or
               //execution will fall through to the next case
    default:
        showWittyErrorPage(); //I don't have an example.
                              //That's an error.
                              //That's all I know.
}

//This translates directly to:
if(requestedPage == "#home") {
    showHomePage();
} else if(requestedPage == "#about") {
    showAboutPage();
} else {
    showWittyErrorPage();
}
```

The ternary operator, on the other hand, is not as intuitive. The general format is

```
(expression) ? ifTrue : ifFalse
```

Example 13 – The *Ternary* Operator

```
var isRealisticExample = false;

console.log("This is " +
  (isRealisticExample ? "a " : "an un") +
  "realistic example."); //"This is an unrealistic example."
```

Loops

There are three types of loops in JavaScript: for, while, and do while; the last type is rarely spotted in the wild, however.

The first, and most often used loop, the *for loop*, actually comes in two different flavors: vanilla and for...in (not quite something I'd eat on a cone). Your ordinary `for` loop takes the form

```
for(Variable; Test; Iterator) { /*do stuff*/ }
or, most commonly
for(var i = 0; i < someArray.length; i++) { /*do stuff*/ }
```

where Number is the variable on which to iterate. Test is an expression, evaluated before each iteration that, if false, ends the loop. Iterator is an expression that is run at the end of each iteration.

The for...in loop is like a for...each loop that iterates over the properties of an object.

Example 14 – Syntax for for

```
var problems = [];
problems[99] = "too lazy to fill array";

for(var i = 0; i < problems.length; i++) {
    if(problems[i] === "can't use profanity in lecture") {
        console.log("I feel you, bro");
        break; //Exits the loop
    }
}
if(i === problems.length) { //the loop finished/didn't break
    console.log("I've got 99 problems.");
    console.log("Most of them are undefined, though...")
}
```

Example 15 – An example to make for...in less foreign

```
var arrayObject = [];
arrayObject.push(1);
arrayObject[-1] = "wtf?";
arrayObject["-1"] = "stahp pls"; //Overwrites "wtf?"

for(var key in arrayObject) {
    console.log(key);
}
//Outputs: "0", "-1"
```

While loops are a bit more straightforward. They only have the Test component and generally look like this:

```
while(Test) { /*stuff*/ }
```

Do...while loops are similar except that their body is run at least once.

If you feel like all this information has thrown you through a loop, you should look at some examples while you wait.

Example 16 – Whiling the time away

```
while(true) {
    yoloSwag();
}
cureCancer();
//To get to the magical land of all numbers big and small,
//go down this road for infinity and then make a left
```

Example 17 – A real life do...while!

```
var reasons = 0;
do {
    reasons++;
} while (false === true)

console.log("I can think of " +
             reasons + " reason to use do...while loops");
//"I can think of 1 reason to use do...while loops";
```

Wrap-up

That was quite a bit of information. Congratulations on making it all the way through. To commemmorate this great acheivement, I would like to offer you this short reference of some of the more important constructs.

To create a new variable, use:

```
var variableName = value; //Can be any object
variableName = value; //Global variable
```

To access the property of an object, use:

```
theObject.propertyName  //key must be single word String
theObject[propertyName] //key can be any String
```

To create a new function:

```
function aFunction([arguments]) { //can be prototyped
    /*code*/
    return value; //optional
}

//can not be prototyped
var anonymous = function() { alert('pass me around!'); }
```

To create a new object using a function:

```
var newObject = new TheFunction([arguments]);
```

To add a property to all objects constructed using a function:

```
TheFunction.prototype.newProperty = value;
//accessed using theObject.newProperty
```