Introduction to Computer Science I
COMP 2406A – Winter 2020

Introduction to Javascript

Dave McKenney
david.mckenney@carleton.ca

Learning Outcomes

by the End of this Lecture, Students that have Completed the Reading Assignment and Review Questions should be Able to:

Use Javascript to do basic programming

Understand variable scoping in Javascript

Understand functions as first class objects

Javascript

Javascript is one of the core technologies of the web

Allows for dynamic web pages

Also now used for server-side programming

Javascript is...

Javascript is... interesting

Javascript is… interesting

It will let you do a lot of things, even if those things don’t make sense

It is easy to miss mistakes you made

Javascript is... interesting

There are many subtle rules that can affect how your program operates

Examples: type coercion, automatic semicolons

Javascript is… interesting

It will let you apply some terrible programming practices

(just because you can, doesn't mean you should)

Javascript is… interesting

Javascript itself does not have general input/output support for network, storage, etc.

This can be an advantage for us, as we will see

Javascript “Hello World!”

Basic output in Javascript:

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

Javascript Execution Environments

The Javascript we write will be executed in two main environments:

Node.js

Chrome browser

Running Javascript code in Node.js is a lot like running other programs from the command line

- `node some_file.js`

01-ex1-hello_word.js example

Javascript in the Browser

Javascript within HTML must be included between the script tags:

<script> ...Javascript here... </script>

Alternatively, you can load an external script:

<script src="01-ex1-helloworld.js"></script>

Your scripts can be included anywhere in the file

Loading scripts at the top vs. the bottom of HTML

What is the difference?

Javascript User Input

Often, the next thing to learn is text user input

We will skip this. Why?

Javascript Variables

Three different keywords for variable creation:

let

const

var

Different scoping rules apply

Javascript Variables

let x = 5;

x's scope is the block it is defined in

Note: block vs function

const x = 5;

x's scope is the block it is defined in (same as 'let')

Note: x cannot be re-assigned

Javascript Variables

var x = 5;

x's scope is the function it is defined in

Note: no block scope

General advice: don't use var anymore

Javascript Variables

Hoisting: Javascript moves some declarations (variables AND functions) to the top of their scope

This means you can actually use a variable before it is defined:

```
x = 5;
var x;
```

(note: initialization of variables is NOT hoisted)

Javascript scoping example code

Javascript Variables

Variables declared with var and function definitions are hoisted

Variables declared with let/const are not hoisted

Javascript scoping example code

Number – integers and floating points

```
let x = 5;
let y = 10.61;
let z = Infinity;
```

Infinity, -Infinity, NaN

String – textual data

Three ways of specifying a string value:

```
let x = “Hello World”;
let y = ‘Hello World’;
```

These two behave identically

String – textual data

Three ways of specifying a string value:

Backticks allow multi-line strings and templating:

```
let x = `Hello ${name}`;
```

Inserts value of name variable into the string

Javascript Data Types

Boolean – True/False

null – nothing/empty

undefined – declared but value not assigned

Javascript Data Types

Object – used to store collections of data and more complex entities

Key/value pairs

Keys referred to as properties of the object

Values can be primitives, functions, objects, etc.

Javascript Data Types

```
let x = {} //creates an empty object
x.someProp = 5 //sets someProp property to value 5
x["space prop"] = 3 //sets "space prop" to value 3
```

Access values in same way

Can also initialize key/values in declaration:

```
let x = { prop1 : “word”, prop2 : 5 };
```

Accessing a non-existent property gives you undefined

```
let x = {};
console.log(x.someProp); //undefined
```

You can get all properties of an object:

Object.keys(someObject)

And delete a property:
delete someObject.someProperty

Object values are references

This has important implications…

(see 01-ex3-javascript-objects.js)

Javascript Data Types

JS objects look similar but are not the same as JSON

JSON is a string representation of a JS object

JSON.parse(someString) will give you a Javascript object created from someString

JSON.stringify(someObject) will give you a plain text representation of someObject

Type Coercion

Javascript performs implicit type coercion

For example:

1 + “2” = “12”

“10”/5 = 2

1 == true → true

Summary table: https://dorey.github.io/JavaScript-Equality-Table/

Javascript type coercion example code

Type Coercion

Automatic type coercion can be a problem when comparing values

Example: “3” == 3 → true

Whether this is what you want depends on the application

Type Coercion

Luckily, Javascript has the '===' operator (and !==)

Compares two values WITHOUT type coercion

Tip: only use === and !===, be aware of what data types you are using, typecast if necessary

Type Coercion

The implicit type coercion can be useful too:

```
if(myObject.someProp){
    //code to execute if object has someProp
}
```

Why is this useful? Why is this dangerous?

‘Falsy’ Values

There are specific values that are considered ‘false’:

1. **false (the Boolean value)**
2. **0 (the number 0)**
3. **“” (the empty string)**
4. **null**
5. **undefined**
6. **NaN**

Ifs/Loops

If statements and loops work just like other languages, but don't forget 'let' when creating a loop variable…

Leaving out let will create a non-block-scope variable that may 'clobber' one of your other variables

Functions

Understanding functions will be critical when coding in Javascript and Node.js

Javascript functions are ‘first class functions’- they can be assigned to variables, passed as arguments to other functions, etc.

This will be important when we look at asynchronous functions

Multiple ways to create functions:

```
function square(x) {
    return x * x;
}
```

Creates a function called square

Remember – this is hoisted (function declaration)

Multiple ways to create functions:

```
const square = function(x) {
    return x * x;
}
```

Creates a function and binds it to a variable

(could also be passed as argument, etc.)

This is NOT hoisted – function expression

Multiple ways to create functions:

```
const square = (x)  => {
        return x * x;
}
```

A shorthand way of defining functions
(input(s) => body/output)

We will also see later that you can create 'anonymous functions' that do not have a name

We will use frequently when doing asynchronous operations

You can pass extra arguments to functions (they are ignored)

square(2, 1, “hello”, {}) still gives you 4

You can also pass too few arguments (others are undefined, or given default values)

Good: gives you some flexibility

Bad: easy to make a mistake and not realize

Arrays

```
let listOfNumbers = [2, 3, 5, 7, 11];
console.log(listOfNumbers[2]);// → 5
console.log(listOfNumbers[0]); // → 2
console.log(listOfNumbers[2 - 1]); // → 3
```

Just like arrays/lists in many languages

Arrays

When accessing an index, you are actually accessing a property of the array object

Since you can't use numbers with dot notation, like:
arr.1 = 15

You have to use []:
arr[1] = 15

Higher Order Functions

Arrays have some useful higher order functions (that is, functions that accept functions as input)

Examples: filter/map/reduce

See 01-ex5-higher-order-functions.js

Synchronous vs. Asynchronous

Synchronous: executed in sequence – the current step finishes before the next step is started (i.e., likely what you are used to)

Asynchronous: the next step can start BEFORE the current step finishes (order of completion is not guaranteed)

We will talk a lot more about this later, as it is an important concept in web applications

Synchronous vs. Asynchronous

Javascript is single-threaded (i.e., only one instruction can be executed at once, technically)

However, asynchronous functions do exist – generally for input/output operations

For example, setTimeout and setInterval functions (see 01-ex6-async-examples.js)

Some important notes:

The time periods are not guaranteed to be exact

A timeout will not execute until the current block has finished executing (single-threaded!)

All callbacks are handled by the ‘event loop’

Synchronous vs. Asynchronous

We will talk more about asynchronous functions, callbacks, event loops, etc. in the future

Great explanation:
https://blog.sessionstack.com/how-javascript-works-event-loop-and-the-rise-of-async-programming-5-ways-to-better-coding-with-2f077c4438b5

Closures

Functions create their own scope – this is why we can have local variables

Inner functions have access to the scope of their outer function(s) – lexical scoping

These inner functions can access the scope even after the outer functions have finished/returned…

(the scope is defined as the code is written, not when it is run)

Closures

So what does this mean? We can do things like this:

```
function foo() {  // 'scope of foo' aka lexical scope for bar
   let memory = 'hello closure';
   return function bar() {
      console.log(memory);
   }
}

// returns the bar function and assigns it to 'closure';
const closure = foo();
closure(); // hello closure
```

Example from: https://astronautweb.co/javascript-lexical-scope/

Closures

Closures can be used to create a ‘function factory’

This allows you to easily create many functions that operate in a similar way

See 01-ex7-closure-example.js

Closures

Closures can also be used for data privacy

You can define Javascript objects with encapsulated state

See 01-ex7-closure-example.js

So now you should…

Go forth and practice!

Figure out the syntax, write/test/debug some basic programs

So now you should…

Read: https://eloquentjavascript.net/ Chapters 1-4

Continue to work on HTML/CSS/Javascript tutorials from w3schools website

Tutorial #1 will be posted tomorrow