Learning Sources
Eloquent JavaScript (3rd ed) ← Primary
MDN Web Docs – MDN Web Docs – Javascript ← Secondary
JS Dev Docs ← Reference

Note on order of notes: order of note sections follow "Eloquent JavaScript," structure, with some additions of info from MDN and other sources

Intro to Javascript

Javascript 101
-Key for dynamic (show different things in different circumstances) content updating. Site that doesn't change = static.

-Scripting lang - interpreted at runtime vs compiled. Executed sequentially.

-API - Application Programming Interface. Built on top of core of language. Allow interaction between core code and more intricate systems (ex. Google Maps) by acting as an in between with ready made components available for use.

-Browser API - exists in web browser. Much of these add additional features (ex. Audio API) or provide additional info to sites (ex. Geolocation API)
-Third Party API - must be added to browser. Ex. Twitter API, Google Maps API

-JS runs after HTML & CSS put together
-Separate tabs run in separate execution env for sandbox-like security

-Client-side code will be executes and updates within the browser

Script Element
-*<script></script>* - element contains JS code w/i HTML doc. Similar to css *<style>* elm. Internal JS. Avoid using as inefficient and messy.

-*<script src="sourceLocation.js" asysnc>* - for linking to external JS source file. *asynch* tells browser to keep loading HTML past JS, so all HTML loaded before JS runs (prevents errors)

-*asynch* vs *defer* scripts - *asynch* will exe as soon as finished downloading. If multi scripts, could exe in various orders. Use when scripts run fine independently. *defer* scripts load in order of appearance in code and wait to exe until previous script loaded. Use when scripts have inter-dependencies.

Debugging Basics
-Javascript dev console in Firefox will display syntax errors from code and specify line of error. Node console also does this.
-JS = case sensitive
-*null* - no value
-*undefined* – has value, but essentially escape value

Values, Types & Operators

Numbers and Arithmetic Operators

General prog lang number types:
-int - whole num
-float (floating point) - decimal
-double - higher precision decimals
-binary - 0, 1
-octal - base 8 number
-hexadecimal - base 16 num, 0-9 then a-f in each column

-JS only uses *number* data type, specified by *var, let,* or *const*, so do not need to specify above. All numbers = 64 bits in JS. Note: 64 bits = high precision, but still not infinite.

-Can shorthand numbers via *1.234e8*, etc.

-To see what data type item is, run *typeof dataName;* on. *typeof* is a unary operator.

-Typical arithmetic operators *+, -, /, *, %* (modulo, aka mod, ie remainder of), ordered by typical order of ops via parentheses.
-increment: ++
-decrement: --

-Special numbers: *Infinity, -Infinity, NaN* (Not a Number (ex. *0 / 0 = NaN*))

Assignment Operators
x+= 4 shorthand for *x = x + 4;*
-Can also do *-=, *=, /=, --,or ++*

Comparison Operators
>, <, >=, <=   == or ===   //equal to. Can be used on strings.   !== //not equal to
-Return boolean value
-Note that == compares by identity, not property
- === compares the value of two objects, but only works on primitives, not objects

Logical Operators
&& - AND || - OR   ! - NOT
-NOT example - *if (! (season === "winter"* || *climate === "polar")){…}*

-Ternary Operator - Tests a condition and runs code A if true, B if false. Good if-else shorthand.
    -Pseudo-code: *(condition) ? run this code : run this code instead*

-Order of operations, highest to lowest: comparison operators, *&&,* ||

Strings
-Can set value with single quotes, double quotes, or backticks (`)
-Can set *stringX* to have *StringY* value via: *stringX = stringY;*

-Quote pairs set within quote set will be read as part of string, not triggered as reserved char
-Unclosed quote (ex. ") within string set will confuse to where string ends/starts, so set as with escape char as \"

-<u>New line</u> via escape char: *n*
-<u>Backslash</u> via escape char: \\

-Can <u>concat</u> string values with +, either with string value, name of initialized string, or combo of

-Can <u>convert string</u> (ex. var numString =*'123';*) <u>to number</u> by calling <u>*Number(stringName)*</u> function. Can <u>vice versa</u> with <u>*numName.toString()*</u> call.

-<u>Backtick strings</u> are called "template literals" and can contain substrings enclosed within *${ },* which <u>can contain arithmetic formulas</u> where the computed result is output in the string
    -ex. *`half of 100 is ${100 / 2}`*   //string displays as *half of 100 is 50*

-String length via *string.length;*

-return specific char - *stringName[#]* where # is position of char in string. 0 oriented (first char at 0, not 1)

-*string.toLowerCase()*   *string.toUpperCase()*

Auto Type Conversion
-Since JS only contains a few types, will try and auto-convert to make type fit with method (ex. *"5" * 2* outputs *10*). Aka "type coercion."

-Good for debugging. To see if value has real value, compare == *null*. If == *null*, error. Note: *Null == undefined*.

Program Structure

-<u>Expression</u> – fragment of code that produces a value
-<u>Statement</u> – a complete expression. Basic statements ended with *;* (though technically not needed, but more error prone)
-<u>Environment</u> – The collection of variables and their values when a program is running

-<u>Side effect</u> – change that occurs in program flow as a result of function, statement, etc.. Ex. function displays text box.

-<u>Function</u> – A named section of a program that performs a specific task when executed. Optionally, take arguments and output return values.
    -Declaration syntax: *function(let nameY) { … }*
    -Call syntax: *function(letArgX);*

-<u>Block</u> – any number of statements grouped into a single statement with { } braces

-<u>Declaration</u> - states type and name
-<u>Assignment</u> - assigning a new value to a variable
-Initialization - assignment done during declaration

Variables
-Aka "bindings". A value bound to an identifier.

-Declaration: *type name;*
-Initialization: *type name = value;*
-Assignment: *name = value;*
-String values in quotes

-*var* variable type can be declared w/ same name multiple times, *let* variable cannot. *var* scope, when in function, is function block, while *let* is only accessible to inner inclosing block.
 -ex. *var* inside a *for* loop inside a *function* would be visible outside of loop. *let* would not.

-Can declare multiple variables at once by stating type, then separating names with commas
 -ex. *let one = 1, two = 2;*

-JS is dynamically typed lang - *var* and *let* can store many data types (numbers, strings, boolean, arrays, objects, etc.)

-Naming conventions: 0-9, a-z, A-Z only. No underscore or number as first char. Only $ and _ special chars allowed. capitalizeLikeThis. Constants named in all caps (ex. *CONSTANTVAR*).

-*const* - immutable variable

-Primitives (simple data type w/ no additional properties/methods) - *string, number, boolean, undefined*

***Console.log* Function**
-Outputs to console (ex. *node.js* console, browser console, etc.). Not displayed on site.
-ex. *console.log("Show this Text");*

Conditional Statements
if (condition) {…}
else if (condition) {…}
else {…}

-Can shorthand a boolean variable by just calling *if(varName)*, which will return T/F
-If only one expression following *if(condition)*, etc., do not need braces
-Can nest if-else statements

-Switch - Alternative to lengthy if-else statements. Takes in value or expression as input and runs through cases until choice matches:

```
        switch (expressionOrValue) {
            case value1:
               run this code;
              break;
             case valueN:
               run this code instead;
              break;
            default:
              actually, just run this code;
}
```

Loops
while(condition) {…} //checks condition, then executes. Could execute 0 times.
do {…} while(condition); //executes, then checks condition. Always executes at least once.

f*or(let i=0; i++; i < 10){...}* *//*etc. Even if omit loop arg(s), still include semicolons

break – can break out of loop by calling *break;* statement

Comments
//comment

/*Multi-line
comment*/

Functions

Definition
-In JS functions are values. If want to define and bind a function like one would with a *var,* etc, name goes on left side of definition, and binding is denoted as a function via keyword *function*
    -ex. *const mathFunction = function(x) {...};* //note semicolon at end, since definition

-Parameters in function only given a name, not a type
-Parameter at definition = argument when value passed as param during runtime

-Return specified via *return x;*

Scope
-Global if outside function, local if inside function
-*let* and *const* only in scope within their containing { } block (use *var* if need to define variable in loop that can read outside of loops, etc.

Functions as Values
-Can pass functions to other functions via assigned name
-If function not *const* can give function a new definition by redefining function

Declaration
-Functions can also be declared in a more traditional manner via, *function functionName( ){…}*
-When a function declared, instead of defined, function out of normal top-to-bottom flow and thus can be called before declaration, as if lang was compiled

Arrow Function
-Can also define a function in a manner similar to piping in unix, with =>
-Syntax: *const functionName = (x, y) => {...};*
-Flow: parameters sent to {...} with =>, which then returns to bound function
-If only one parameter, can omit ( ) surrounding param. If no params, use (). If only one expression contained within function, can also omit { }.
-Less verbose way of writing function expresses

Call Stack
-Current place where code is executing sits on top of call stack. When program swtiches to another point in the code (ex. functionA calls to functionB which is lower in the code), current point of execution is stored at top of call stack. When execution of functionB is done, call stack pops and returns to point where it was prior to moving to functionB.
-Call stack stored in memory. If stack grows to large, stack overflow will occur

Optional Arguments
-JS <u>does not error if you pass an extra argument </u>to a function, even if the function was not set to take that argument during deceleration.
    -ex. call *functionName(10, 20)*, but function was declared as *function functionName(x){…}*

-Also <u>does not error if pass too few args</u>, and simply values missing args as *undefined*. If place = *valueX* after arg during function declaration, that value will be default if arg not defined, instead of *undefined.*

Closure
-Allows you to use a name for a variable in local scope, then use the same name again later, once that variable name is out of scope

-ex. name *let dog* in for loop in function, then name another variable outside of the for loop in the same function, after the for loop block

Recursion
-When a function calls itself
-Make sure there is an end condition specified so the program doesn't enter (((infinite recursion))) and stack overflow
-Note that using simple for, etc. loops is often significantly faster during runtime than making recursive calls. Upside is less wordy, more elegant code.
-Useful when traveling through branches, such as search traversals

-Remember that if returning a value from base call of recursively called function, all conditionals, etc. in the statement must also return a value, otherwise function will return *undefined*, as return chain will not exist in recursive calls of function

Data Structures: Objects & Arrays

Properties
-Declared values <u>stored within objects</u> (ex. *string.length)*

-Reference *object.propertyName* <u>to access property</u> by name

-If accessing a property that has an <u>atypical binding name</u> (ex. numerical binding names such as an array index or binding name with spaces), access via <u>*object["binding name"]*</u>

-ex. *objectA.Object.keys(objectA)[4]* ...
//Object.keys() returns an array of property names for an object. Here, *objectA* is referencing the name of its 4th property. Since the name is returned from the method *Object.keys()*, it needs to be referenced via a []

Methods
-Properties that hold function values
    -ex. *string.toUpperCase()*

-Call via: *object.Functionname(optionalArgs);*

Arrays

-In JS, linear arrangement data structure where you can add to either end and remove from the end (highest index). Note, this gives it stack and queue like functionalities.

-Initialization - *let arrayName = [data1, data2, dataN];*
    -In JS, arrays can hold mixed data types, as a *let, var,* etc. could be a number, string, object, etc.

-Access item via *arrayName[#]*, where # is index of item. Assign val via *arrayName[#] = val;*
-Multi-dimensional access via: *arrayName[#][#],* etc.
-As all data structures, first index starts at 0

-length - *array.length*

-Add to end of array - *array.push(data, optionalDataN);*   *//returns new array length*
-Remove from end of array from array – *array.pop()*
-Can create equivalent of stack (LIFO) with above

-Tell if array includes value. Boolean. - *arrayName.includes(valueX)*

Array Loops
-Instead of looping through array in traditional manner, by calling *arrayName[i]* via incrementing *for* loop counter, can do:
    for (let indexName of objectName) {…}   *//let* could also be *const, var,* etc

-Note that you are actually assigning the indexes of the array a name with index*Name.* This is equal to

```
for(let i = 0; ........){
    let someName = arrayX[i];        //could then reference someName.someProperty;
}
```

-Similar to *foreach* loop in Java, which states "for each x in y, do ..."

Objects
-Collection of properties and methods in an contained group referenced by a specific object name (global) or object instance name (local or global)

-Definition: *let objectName = { propertiesNameX: value,*
propertyNameY value
"property name z": "the value"};

-Primitives types are copied by value, reference types are copied by reference

-Note that core objects, such as *Math* and *Object* are named with uppercase

-Property assignment (outside of object) *objectName.propertyName = value;*

-Can remove property from object via unary operator *delete objectName.propertyName;*   Note this doesn't just set the value to undefined, but actually removes the property from the object.

-Can tell if property exists in object (referenced via name) via binary *in* operator: *"propertyName" in objectName;*   //note quotes around property name

-List object properties - *Object.keys(objectName);* //Object is a per-existing obj in JS core

-Copy all properties of objectX to objectY - *Object.assign(objectX, objectY)*
 -if both objects have property of same name, Y value will overwrite X value

-Object example:

```
function addEntry(tasks, complete){
    toDoList.push( {tasks, complete} );         //pushes object containing tasks & complete
}

addEntry( {["work", "study", "clean room", "call Bob"], true},
          { ["work", "water plant", "bank"], false} );
```

//pushes two entries, where *task* is an array of strings and *complete* is boolean

Mutability

-Even if objects contain properties with the same names, each object references a different property, as the objects are independent of one another, as long as object does not reference another existing object via objectA = objectB, in which case a change in one would reflect in both.
 -A comparison of *objectA* and *objectB* with === in this case would return *false*, though, since it compares by identity, not content

Advanced Arrays

-Add to start of array *array.unshift(data, optionalDataN);* *//returns new array length*
-Remove from end of array - *array.shift()*
-Can create queue (FIFO) with above

-*arrayName.indexOf(valueX)* - searches from start (index 0) to end of array and returns index of value, if present. Put value in qoutes if has spaces. Returns -1 if not found.
-Can do same, but start at end of array via *arrayName.lastIndexOf(valueX)*
-Both also take a second argument for where to start searching, moving in either ascending or descending search order from that position

-*arrayName.slice(indexStart#, optionalIndexEnd#)* - copies sub-array out of array. If don't specify end char, slices from start until end of array.

-Concat arrays with *arrayA.concat(arrayB)* - Can also define an array for *arrayB* (*[valueX, valueY, ...])* instead of passing existing array name.
 -Can also pass arguments that are not arrays, and will concat to end of array

String Methods

-Strings have *indexOf* and *slice* methods, like arrays:
 - *stringA.indexOf("char(s)")* //could search for *"search"*, etc.
 - *stringA.slice(startChar, optionalEndChar)*

- *stringB.trim()* - Remove whitespace (spaces, newlines, tabs, etc.) from start and end with

- *string.replace('originalSubString', 'newSubstring')*

- *padStart(timesToPad, "padChar"*) - <u>adds *padChar* x number of times to start</u> of string

- *stringX.Split("char(s)")* - <u>splits string</u> into array of strings at every occurrence of *char(s)*

- *stringX = arrayName.join('breakChar(s)')* - <u>array to string</u> where array values are separated in string by *breakChar(s)*

- *stringX.repeat(#ToRepeat)* - <u>repeats string</u> # of times

Rest Parameters
-Can set a function to <u>accept any number of args</u> by placing three periods before last argument:
    - *function functionName(...lastArg) { …}*
-... is a "rest parameter," and bound to an array that holds it's args

-Can also call a function with an array argument using rest parameter, where specifying rest param <u>spreads out the array so each array element is passed to the function as an individual arg</u>
    -ex. *printElements(...myArray);*

-If use rest param <u>in array assignment, will spread out rest array inside array added to</u>
    -ex. *let firstArray = {"This", "is", "array"]*
        let secondArray = {"Dog", ...firstArray, "Cat"
        //secondArray contains *"Dog", "This", "is", "array", "Cat*

The Math Object
-*Math.max(args)* - returns highest num of args
-*Math.min(args)*
-*Math.sqrt(arg)* - square root

-Trig Math functions - *.cos* *.sin* (sine) *.tan* (tangent) *.pi* *.acos* *.asin* *.atan*

-*Math.random()* - returns pseudo-random num between 0 (inclusive) and 1 (exclusive)

-*Math.round(wholeNumOrExpression)* - rounds decimal to nearest whole num
-*Math.floor(wholeNumOrExpression)* - rounds decimal down to nearest whole num
-*Math.ceiling(wholeNumOrExpression)* - rounds decimal up to nearest whole num

-*Math.abs(num)* - returns absolute val of num (negates negatives to become positives)

JSON
-Objects and arrays stored in memory during runtime, where data mapped to mem addresses
-JSON <u>serializes memory that holds data into a flat description.</u>

-In JSON, property names must be double quoted and only simple expressions (no function calls, bindings, etc.) are allowed. Also no comments

```
-Ex {
        "man": false,
        "names": ["kelly", "diana", "maria", "karen"]
    }
```
-To <u>convert data to and from JSON</u>, use *JSON.stringify* and *JSON.parse*, where when converting to, you pass it data such as above example.
    -ex. *JSON.parse(cities).populations;*

Objects & Flow
-In the following code, a linked list is created. Note that the list element creates a link property that references itself. If the first iteration starts at 3, the list property holds itself, which at the time off assignment, is null. On the next iteration, initialization has occurred, and list now is 3. When assignment occurs, it thus contains list 3, while the value is set to 2. Another iteration passes and it sets the list to the current value of 2, while value changes to 1, etc.

```
function arrayToList(theArray){
        let list;

        for(let i = array.length-1; i >= 0; i--){
                list= {value: theArray[i],
                        link: list
                };
        }
        return list;
}
```