JavaScript ES9 & Beyond

Notes open for creative commons use by Kyle Miskell @ developer blog: https://kylemiskell.com, github: SmilingStallman*, email: klabweb@tuta.io*

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 \”

-New line via escape char: *n*
-Backslash via escape char: \\

-Can concat string values with +, either with string value, name of initialized string, or combo of

-Can convert string (ex. var numString =*‘123’;*) to number by calling *Number(stringName)* function. Can vice versa with *numName.toString()* call.

-Backtick strings are called “template literals” and can contain substrings enclosed within *${ },* which can contain arithmetic formulas 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

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

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

-Function – 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);*

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

-Declaration - states type and name
-Assignment - 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 enclosing 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
-Less verbose way of writing functions

-Syntax: *(x, y) => {...};*
    -If only one param, ( ) are optional
    -If returning expression, no brackets: *(x, y) => expression*
    -If want to return to variable *let varName = (x,y) => {...}*

-Flow: parameters sent to {...} with =>, which then returns

Call Stack
-Current place where code is executing sits on top of call stack. When program switches 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 does not error if you pass an extra argument 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 does not error if pass too few args, 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 stored within objects (ex. *string.length)*

-Reference *object.propertyName* to access property by name

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

-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;
}
```

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 start 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 quotes 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"*) - adds *padChar* x number of times to start of string

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

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

- *stringX.repeat(#ToRepeat)* - repeats string # of times

Rest Parameters
-Can set a function to accept any number of args 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 spreads out the array so each array element is passed to the function as an individual arg
-ex. *printElements(...myArray);*

-If use rest param in array assignment, will spread out rest array inside array added to
-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)

Destructuring
-Allows you to unpack elements or properties from arrays, objects, etc. into distinct variables by using [] or {} as lefthand assignment operator

-With objects, reference by property names:
const theObject = {a: 1, b: 2, c: 3};
const {a} = theObject //will print 1
const {a, c} = theObject //a will print 1, c will print 3

-Can give different names then name of object property via
const {newName : a} = theObject //newName will print 1

-With arrays, works via indexes
const theArray = [1, 2, 3, 4];

const [a, b] = theArray; //a prints 1, b prints 2

-To skip index in array, use extra commas, where each additional extra comma is an index
-ex. *const[, a, , b] = theArray; //a prints 2, b prints 4*

-If use rest parameter before variable name destructuring too, remainder of array will go into rest
-ex. *const [a, b, ...rest] //a prints 1, b prints 2, rest is an array with 3, 4*

-Note, can also destructure on definition
ex. *const [a, b, ...rest] = getBigArray();*

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

-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 convert data to and from JSON, 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;
}
```

High-Order Functions

Abstraction in Functions
-Taking a complex problem, likely with long wordy code, and breaking it down into smaller parts, which can then be used to solved them same problem in a less wordy, "easier to grasp when reading the code," way

-Concepts such as breaking down large functions into smaller functions, sending functions objects to actions on, defining a function in the call instead of creating a bind for it prior then sending, etc.

High-Order Functions
-Functions that operate on other functions, either by taking them as args or returning them

-Pure function - does not modify the arguments that it is given. No side effects. Less likely to cause hard to decipher bugs due to unexpected side effects.
-Can create more pure functions by having functions that are known not to pure operate as separate functions, then called on by pure functions.

Transforming with Map
-Pass a function to act on array elements and return new mapped elements:

```
function map(array, transform){          //transform is a function
    let mapped = [];
    for (let element of array){
        mapped.push(transform(element));      //runs transform() on elm, then pushes to array
    }
    return mapped;
}
```

Example with Comparison Operators

```
function inBetween(year){
    for(let person of people) {
        if(person.checkYear(([born, died]) =>
            { return year >= born && year < died;} ))
        return person;
    }
}
```

//Runs checkYear() function that person has as a method and passes it an array with *born* and *died* years to function that returns *person* object if that person's year falls between born and died

Summarizing with Reduce
-Returns a single value generated from performing the passed computation function on each item in array (ex. a some function for numbers) starting with *firstElement*

-arrayName.reduce((accumulator, currentValue) => {return ...}, intialValue);

-Built in function for arrays in JS. Takes two arguments:
1) function: *(accumulator, currentValue) => {return ...}*
2) initialValue.

-currentValue is used by *reduce* to store the current index's value on the current iteration
-initialValue is the first value for *accumulator* to use on the first iteration

-reduce runs through the array and for each item in it, returns the function output defined in ..., which usually involves performing an action on the *accumulator* and *currentValue*

-Sum example:

```
let numbers = [0, 4, 2, -8];

let sum = numbers.reduce((accumulator, currentValue) =>
            accumulator + currentValue) , 0);

//accumaltor initial value = 0. On each run adds index value, ie currentValue to accumulator
```

More Built in Array High Order Functions
-Numerous like *slice(), foreach(), pop(), includes(), indexOf()* etc. already discussed

-*arrayName.filter(testFunction)* - Returns an array of values from *arrayName* that pass the test specified in *testFunction*

-*arrayName.map(transformFunction)* - Returns an array of values after applying the transform function

-*arrayName.some(testFunction)* - Returns true if any elements in array pass test specified by passed test function

-*arrayName.every(testFunction)* - Returns true if every element in array passes test specified by passed test function

-*arrayName.reverse()* - Reverses the order of elements (front index becomes back index, etc.) of array

Objects and OOP

Encapsulation
-Breaking programs into smaller pieces where each piece manages its own state

-By localizing functions to objects, the inner workings of the objects do not need to be known and can be accessed via abstract interfaces, which remain consistent, even if the inner workings change. Interfaces = public, inner workings = private.

-Since JS does not include native *public* and *private* keywords (sigh...), so common to ghetto-rig it by name private functions starting with _

Methods
-Object properties that hold function values
-Name an object property with dot

```
-ex. let rabbit ={}                         //object definition
     rabbit.speak = function(arg){...}      //method deceleration
```

-Can refer to method from within itself via *methodName.propertyName*

```
-ex. function foo() {
         foo.count = 4
     }
```

this
-A runtime binding made when a function is invoked, where either
a) if made within a method call, *this* references the object that is executing the current object.

b) if made within a global function call, *this* references the global object (ex. browser window)

-Ex. *const video = {*
title: "Nightslayer III",
play(){ console.log(this.title}; }
}
video.play();
//output is "Nightslayer III"

-*this.type* - when called from inside object, returns object name

Protoypes
-All JS objects have a property that points to a prototype object, which is the object that type of object was based off of. Any method or property the prototype has, the objects cloned from it will also have.
-If the prototype is changed these methods/properties will also changed for the objects that were created from it.
-Objects created as clones from prototype can override prototype methods/properties w/o changes reflecting in prototype

-If request is made on object for a property it doesn't have, it will check to see if it's prototype object has it and return it instead

-Some core JS prototypes: *Object.prototype, Function.prototype, Array.prototype*
-*Object.getPrototypeOf(objectName)* - returns prototype of object

Prototype Constructors
-Class - blueprint containing all methods and properties all objects (instances) of that class will have. JavaScript uses prototypes instead

-Create object of specific prototype: *Object.create(prototypeName)*
-ex. *let dog = Object.create(protoDog);*
//dog is created as an object cloned from *protoDog* and thus shares methods/properties of

-Constructor - function that creates an instance of a prototype with the proper instance-specific properties
-ex. *function makeRabbit(type){*
let rabbit = Object.create(ProtoRabbit);
rabbit.type = type;
return rabbit;
}
//object *rabbit* is created as a clone of *protoRabbit* and given additional passed *type* value

-Can also construct a new object by calling the *new* operator which creates a new instance of the prototype specified to the right of it. Usually call *new* on constructor methods:

-ex. *function Car(type, model) {* //constructors are capitalized
this.type = type;
this.model = model
}

let usedCar = new Car("sedan", "civic";

console.log(usedCar.type);

-All user created objects inherit a *prototype* property, which is the *prototype* they were derived from and can be viewed via *Object.getPrototype(objectName)*. When create instance with *new*, this value is set to object it is cloned from

-ex. (continued)
Object.getPrototype(Car); *//returns [Function]*
Object.getPrototype(usedCar); *//returns Car {}*

Class Notation
-*Class* keyword defines a class type, so don't need to use *const*. Create *constructor()* method inside class, to create instances of class with via *new*, etc.

-ex. *class Car{* //also capitalize prototype names
constructor(...args) { //leave as *"constructor"* as methods as *new* calls this name
this.bindA = argA;
}
//methods here
}

let newCar = new Car("shiny"); //calls *constructor()* method and creates Car clone

-Note that classes themselves can only hold methods, not properties. Properties must thus be specified by methods, such as in the *this.type* constructor example above.

-Could also create new class like: *let object = new class {…}*

-Can create object with no prototype by calling *Object.creat(null)* when creating object. Note, will not include *Object* prototype methods, like *toString()*

Overriding Derived Properties
-When override property in instance that also exists in prototype, prototype property simply ignored and instance property referenced for *this* instances instead. Only overridden in instance and prototype and other instances created from prototype unaffected.

-Allows you to make a more specialized version of a class derived from a more general prototype

Maps
-Data structure that associates keys with values

-Can check if property name exist in object via *in* operator, which is called on object name
-ex. (continued)
return "type" in Car; //returns true

-Maps allows similar functionality, via *Map* class, where object is created via *let myMap = new Map();* and then interact with via:

-myMap.set("myKey", myValue)
-myMap.get("keyName) - returns associated value
-myMap.has("value") - returns boolean

-Excellent for quickly updating and searching a large set of data

Polymorphism
-The ability to call the same method on different objects and have each of them respond in their own way. One interface may be implemented by multiple objects. In JS, formed by varying functionality of constructors.

Symbols
-JS includes a *symbol()* method that takes a value and associates a unique identifier to that value. This allows you to set symbols as property keys, where their names are actually unique compared to strings. This allows you to avoid name-clashes for properties or provide truly unique object keys.

-Create via *let name = Symbol("value");*
-Note that *value* is not what *Symbol* uses to generate the unique symbol, but simply an optional descriptive String parameter that is show when displaying the Symbol

-Example (cont.):
```
const meltCar = Symbol("meltCar");
Car.[meltCar] = function() {...}; //
```

-Example
```
let stringA = "the;
let stringB = "the"          // stringA == stringB  returns true
let sy8   mbolA = "the";
let symbolB = "the";         //symbolA == symbolB returns false
```

The Iterator Interface
-In JS, an iterator is a pointer for for traversing the elements in a data structure
-Iterable objects are ones that implement the *iterable* interface. Any object that contains a *[Symbol.iterator]* method is an iterable object.
-*iterator* is a part of *Symbol* to give it a truly unique identifier, which you can append to functions to make them implement *iterator*

-*String, Array, TypeArray, Map,* and *Set* all implement *Symbol.iterator. Object* does not.

-By giving a class a function that references [*Symbol.iterator]* and returns an *iterator* object (which is responsible for the iteration logic) you make the class iterable. Then, when a function that uses iteration is called (ex. *for...of)*, that function calls on the class Symbol.iterator method for iteration.

-This interface has a *next()* method to return the next result (which is an object with a *value* property for the next value), and a boolean *done* property for if there are more results or not.

-To make an objet iterable, you would define the object, the link it with the iterator interface:
-ex.
```
let iterable = "IterateMe";
let iterator = iterable[Symbol.iterator]();
console.log(okIterator.next());          //value: "I", done: false
console.log(okIterator.next());          //value: "t", done: false
…
 console.log(okIterator.next());              //value: undefined, done: true
```

-Can also make iterable via shorter syntax:

let okIterator = "OK"[Symbol.iterator]();

-To create a custom iterator class or object, give it a method *[Symbol.iterator]()* {...}, then override the functionality of the *iterator* interface as desired by giving it a *return { next: () => {...}};* method (an iterator) that returns *{value, done}*. *Done* could be implemented by using an element counter, etc..

-*Eloquent JavaScript* did a bad job covering this, imo. Use this resource for a much better description of iterators in JS

Getters, Setters, and Statics
-Getter - return an object property from an object (or a modified version of a property, such as a property with an expression calculated on it). Define by precursing method name with *get.* Allows for proper encapsulation and abstraction.
-ex. *get fahrenheight() {return this...};*

-Setter - sets the value of an objects property (or ""). Define by precursing method name with *set*
-ex. set *fahrenheight() {return this...};*

-Once defined, can then set and get properties from global by calling *objectName.get = ...;*
-ex. *chicago.fahrenheight = 86;*
console.log(chicago.fahrenheight); *//prints 86*

-If set keyword *static* before a method, method is not called on an instance of the class, but the class itself. These methods can thus call the *new* method and the constructor name for the object, and an object of *this* object type will be returned This allows you to created different types of objects using the same constructor, using different *static* methods.

-ex. *class Temperature {*
constructor(celcius) {
this.celcius = celcius;
}

static fromFahrenheight(temp){ //Tempreature.fromFarenheight() returns Temp obj
return new Temperature((temp - 32) / 1.8));
}

Inheritance
-A new class(subclass) that inherits from the old class (superclass) inherits the properties and behavior of the superclass without needing to redefine them

-Define subclass using *extends* keyword:
class NewClass extends OldClass { //constructor, set, etc }

-Call constructor of superclass from within subclass by calling *super* method, which takes same parameters as the superclass constructor. To call a method of the superclass from the subclass, call via *super.methodName()*

-Can override a superclass method for the subclass by giving the subclass method the same name, but giving it different behavior, etc.. Functionality is then modified for subclass, but left unmodified in the superclass.

-Inheritance is controversial, because unlike encapsulation or inheritance, which creates a clear divide between classes, inheritance ties classes together. Thus, do not use it as your first solution, as it can create overly complex webs in object structure and functioning.

The *instanceof* Operator
-Binary operator that returns true if object is derived from a specified class
-Syntax: *className instance of Class*

Bugs & Errors

-Debugging - the process of finding mistakes in code syntax or logic

Strict Mode
If put *"use strict"* at top of file or as first line in function:

I) JS will error if a type is not defined for a variable, instead of the usual behavior, which is to assign the variable as a global variable. Only works if binding does not already exist globally.

II) *this* gets set to undefined in functions that are not methods.

III) it will not allow functions to have parameters of the same name

Types
-Since JS is so weakly typed, useful to put comment before function that specifies what types args should be and what type returned
-Can also use *TypeScript* which is a superset of JS produced by MS that adds stronger typing

Testing
-Can write your own tests or use per-existing suites of tests that output info on when a test fails ("test runners")

Debugging
-Stick some *console.log()* calls at strategic spots to help identify where the error is occurring or the error chain starts
-Browsers contain debuggers that set a breakpoint at a specific line in your code, where the execution the pauses and you can then expect binding values at that current state
-If include *debugger* keyword in code, browser will breakpoint at that spot

Error Propagation
-Make sure to account for user input error, either handling errors and return a special value (ex. *null*) or displaying an error to user. Display often preferred.

Exceptions
-Exception handling - when a program errors and cannot proceed, so it jumps to a place to handle the error, then returns to normal flow

-Exception is thrown by erroring method and caught by error handler. Error handler will the "unwind the stack" of the error, producing the flow of function calls leading to the error origin

-Exceptions in JS very similar to Java, via *try, throw, catch, and finally* :

1) put *throw* statement in function you want to throw error if error occurs:

throw new Error("Error description: " + someValue) //ex. put in user prompt function
2) Call function that may error from within try catch statement, where the function call occurs within *try* and is followed by a *catch(error)* that catches the error if one occurs. If *catch* triggers, error is displayed then program continues from below *catch* statement
3) Optional *finally* can go below catch. Code inside *finally* executes regardless if error is caught or not

-ex.

```
function doSomething(x){
  let value = prompt(value);
  if .... return...;
  if .... return...;
 throw new Error("Error message: " + value);

try {
  doSomething(reeeee);
}
catch (error) {
  console.log("Error: " + error);
}
finally{
   doThisNoMatterWhat(){...};
}
```

-Note that can throw anything (ex. instead of throwing *Error* could throw *FishError*) and the error throw will be of that type

-When *catch* is triggered, stack-trace also occurs and is stored in the *stack* property, which can then be reviewed

-The fewer side effects program functions have, the less likely an exception is to cause problems with the following flow

-To create proper flow, write *try catch* statements in a wrapper function that calls the function you want to pass through the *try/catch* and do the error handling in there

Selective Catching
-If no *catch* statement, program will either execute and error at end or halt, depending on engine running JS. Good for debugging, but terrible for real world use during runtime.

-Cannot selectively catch exceptions in JS. Either catch them all, or don't catch any, so may miss exceptions as a result. Can mimic selective catch by checking exception for value to see if intended error, and handling if so, and otherwise letting pass through unhandled.

Assertions
-Functions set to trigger if an error occurs and throw an error, etc.

Regular Expressions - come back to more later

-Patterns used to match character combos in strings. In JS, the expression is a *RegExp* obj

Creating a RegEx
-Can build with *RegExp()* constructor or define within /.../

let regA = new RegExP("abc"); //standard string backslash rules
let regB = /abc/; //put backslash \ before \ also before + and ?

Testing for Matches
- *someRegEx.test("input")*; //tests to see if *someRegEx* exists inside *input.* returns boolean.

Sets of Chars
-If define pattern within [ ], defining a set. Ex. */[012345]/*. Can be quick range set with hyphen: */[0-9]/*. If run *.test("input)* on set, *test()* returns true if *input* contains any of chars defined in set.

Character Groups
-Match any digit in their group. Ex. *d* matches all digits.
\d - digit \w alphanumeric \s whitespace (newline, tab, space, etc.)
\D - not a digit \W non-alphanumeric \S non-whitespace

-Can include as parts of sets. Ex./*[\d.A]/* //set for any digit, ., or A

-NOT: expressed with ^ inside [] ex. /[^0123]/) //not these num chars

Repeating Parts
-Occurs any number of times > 0: *char+* ex. *d+ABC* //any-num-digitsABC
-Occurs *any number of times including 0: char**
-Occurs zero or one time. Is "optional": *char?* ex. */neighbou?r/*

-Occurs x num of times: *char{x}*
-Occurs x to y num of times: *char{x, y}*

-See cheat sheet for more RegExp selection characters

Grouping Sub-expressions
-Can run tests above on groups of chars by enclosing groups within ( )
-ex. */boo(ho+)+/* //boohooohohooohohooohooo is true

Matching and Groups
-Execute function *someRegEx.exec("input")* - returns object containing array of strings, where each index is substring that matched pattern
-Can call properties on object to see what index of *input* substring starts. *.index* for current index.

-Can also do the same with *"someString".match(/regEx/)*, Also can take a string parameter.

The Date Class
-Date object - *new Date();*
-In JS, months start at 0, but days start at 1
-Arguments are numeric *(yyyy, mm, dd, hr, min, sec, ms)*. Last four args optional. UTC.

-Can take single arg as a millisecond count of the date *new Date(13874929840000);*
-Can get current ms count by calling functions *By creating a* new *Date* object and calling *getTime* on it or *Date.now* function. Also has functions *getFullYear, getMonth, getDate, getHours, getMinutes, getSeconds*

Modules

-Used to avoid big ball of mud program; hard to understand and maintain.
-Module - piece of program that specifies which other pieces it relies and and provides and interface for other modules
-Dependencies - other modules required for moduleABC to function, similar in Linux PM

Packages
-Pretty much the same as Linux
-A chunk of code that can be copied and installed (distributed) which can contain one or more modules and lists its dependencies
-Usually comes with doc on how to purpose and use
-When package update by creators, users can also update package

-Packages installed, updated, etc., via NPM, the Node Package Manager, bundled with Node.js
-Many, many quality frameworks, libraries, packages, etc. available through NPM (ex. REACT)

Improvised Modules
-Since modules were an ES6 addition, makeshift module construction was often done by creating "private" *const* data structures, then returning data from them via a makeshift API

Evaluating Data As Code
-Function constructor - takes list of arg names and function, and runs function on passed args when called. Syntax: *let someFunction = Function("argA, argB, ...", "function statement;");*
-Code thus has isolated scope, wrapped inside function value

-ex. *let plusOne = Function("n", "return n+1;");*
console.log(plusOne(4));

CommonJS
-Used by Node and most Node packages
-Has a *require()* function, which loads a module

-Export modules with *module.exports = someFunctionObjectEtc; or module.exports = {functionA, letB, ...}*

-Once exported, assign to variable via *const someVar = require('./moduleName)*, then can call elements from module

-ex. *function add(a, b) { return a+b;}*
module.exports = add;
*//*some other file
cont add = require('./add');
console.log(add(4, 8));

-Note that interface for module can be a function, such as above example

ECMAScript Modules
-Uses *import* keyword to import parts of module.
-Syntax: *import something from "someModule.js";*
-ex. *import React from "react";*
-Import multiple parts of module via *import {something, somethingElse} from ...;*

-Can then call function names, etc., by name, as if were globally bound function, etc.

-Export in similar manner via *export* keyword
-*export* may appear prior to a function definition, class definition, or binding name to one
  -ex. *export function formatDate(date, format {...};*

  -ex. *export {thisFunction, thatFunction, someObject};*
      //would then import via *import {thisFunction, someObject}* from "*jsModuleFileName";*

-Note that *export* sends out *binding* to function, etc., not a copy of, meaning if definition changes, modules which use module via *import* will see changes

-*default* - if export something with *default* for binding *(*ex. *export default ["Winter", "Summer"];*), used on import if import does not contain braces around around what it is importing
-Braces generally used to import specific parts of module (ex. a function), while *default* imports used to import bulk or standard parts of module

-Can rename bindings of imports via *import {someModule as newName} from "theModule";*, then access methods, etc. via *newName.someMethod()*

-Imports occur before script starts running. Put at top of code for readability.

Building & Bundling
-Bundlers - Tools that put all JS into one file, for faster transfer when page loaded
-Minifier - tools that make the JS file smaller by removing whitespace, shortening bindings, etc.

Module Design
-Follow existing conventions
-Functional programming often faster and more intuitive than stateful OOP programming
-Use proper data structures

Asynchronous Programming - come back to later

-Thread - A running program whose execution may be interleaved with other running programs. Can run synchronous processes across multiple threads for increased speed.

-In asynch, instead of threads, a split in processing occurs, where the program that the split originated keeps running, and the split notifies it when it has completed. Browsers and node both run asynch.

Callbacks
-Callback function - a function passed to another function as an argument
-If function is slow, can pass in callback function, then call function when actions of function passing to complete.
-Ex. pass an anon function to functionX that calls a *done()* method
-Using callbacks this way a bit deprecated after ES6, but good to know for maintaining older code

Promises
-An asynch action that may complete at some point and produce a value. Has ability to notify members when its value is available. Is in either pending, fulfilled, or rejected state.

-Can create promise by calling *Promise.resolve(someObject)*, which wraps *someObject* in a Promise.
-To get result of promise, call *someObject*.then(*value => someFunction{...})*, where *value* is passed as an arg holding the result value.
-*then* also returns a promise which revolves to the value the handling function returns

-Can add multi *then* callbacks to a single promise and will all be called

JS & The Browser

Networks & The Internet
-Network protocol - a style of communication over the network. http, ftp, etc.

-HTTP - hypertext transfer protocol. Receives and sends named packets of data.

-TCP - Transmission Control Protocol. Most communication on internet built on top of it. High level overview: server is listening for clients to start talking to it, with different types of listening on different ports (often with defaults given, ex. SMTP port 25). Listener gets connection request and remote connects via port. Client-sever. HTTP data passes through connection established by TCP.

The Web
-The set of protocols and formats that allow browser access to the net
-Most basic connection = a machine connected to the net with listener on port 80 so other computers can can ask it for document

-http:// - protocol
-thewebsite.com - sever
-/week09-practice.html - path to the document

-Can connect to sever ip via domain name assigned to it via DNS

HTML
Can enclose JS in *<script>...</script>* tag in HTML doc or import via *<script src="./location"></script>*
-If give *<script>* attribute *type="module", can use ES modules*

Sandboxing
-JS not allowed access to files hosted on browser computer and has page only mod rights

The DOM

-JS interacts with DOM as live data structure, meaning dynamic updates
-Dorm forms a tree structure based on parent-child relationships
-Given access to DOM by global *document* object,which has properties referring to*<html>, <body>, <head>, <footer>, , etc.*

- *document.documentElement* - refers to *<html>* tag
` -ex. *let html = document.documentElement;* //for manipulation of *<html>* and children

-JS nodes on tree represent html elements. Nodes have properties, which access CSS & HTML properties

-*nodeType* property - property of JS DOM nodes. ls code number. Elements = 1, text nodes = 3, comments = 8. Will also equal to proper *Node.ELEMENT_NODE*, *Node.TEXT_NODE,* and *Node.COMMENT_NODE* properties.

-Plain JS interaction with DOM is often clunky and messy. Libraries often used for this reason (ex. REACT).

-*someTextNode.nodeValue* - holds text for text node

Important Types
-*Document* - the root *document* object

-*Element* - elements returned by dom API are of type Element. Implement DOM *Element* and *Node* interfaces

-*NodeList* - an array-like object of elements returned by DOM methods that return multiple elements.(ex. *getElementsByTagName()*). Can access elements in via *someNodeList[i]* or *someNodeList.item(i).* Call *Array.from* on if want it to be full *Array*

-*NamedNodeMap* - a map, but elements accessed by name (or index, to allow for enumeration) via *NamedNodeMap.item().*

Moving Through the Tree
Below are properties of DOM nodes
    -ex. *document.documentElement.firstchild*

- *parentNode* - parent node (if any)

-*childNodes* -points to array-like object holding its children elements. Has a .length property. Need full *for* loop with iterator to traverse.

-*children* - points to array-like object holding children, but only type 1 (element) children

-*firstChild, lastChild, previousSibling, nextSibling*

-*nodeType* - ex. *ELEMENT_NODE*

-Moving through DOM often smooth when done with recursion since DOM == tree

-Moving this way can be problematic if DOM structure changes later

Finding Elements
-*Node.TEXT_NODE* created even for white space between nodes in HTML, so traversing by *nth* child can be impractical for many situations

-*document.getElementsByTagName("tagName")* - Gets all descendants of *tagName* for node. Can call on specific children in via *[x]*

- *document.getElementByID("idName")* - single node returned

- *document.getElementsByClassName("className")* - returns all elements of *className*

Changing the Document
-Once have element, can reference html attributes by calling *.attributeName* on item
    -ex. *document.getElementById("checkOutAnchor").href = "/newCheckOut.html"*

-Note that when you reference an element through the DOM, you're doing so via JavaScript APIs. So for example, if you create a *let* that references a specific *<table>*, that *let* inherits from multiple JavaScript APIs, such as *HTMLTableElement* and *Element* to actually interact with, manipulate, delete, add to, etc. that element.

-*someElement.remove()* - Can remove element from DOM by calling on

-*document.createElement("tagName")* - Can create new element of <someTagType>

-*node.appendChild(node)* - appends as last child

-*node.insertBefore(newnode, existingnode)* - inserts newNode before existingNode
    -could call by *node.insertBefore(newNode, node[0])*, etc.

-*node.replaceChild(newNode, nodeToReplace)* - Node to replace must be child of *node* method called on

-Note that a node can exist in only one place in the document, so appending existing node, inserting existing node, etc. will cause it to move to that position, not copy

Creating Nodes
-*document.createTextNode(newNode)* - Can pass a string or an existing node holding a string value:
    -ex. *document.createTextNode(getElementByTagName(theImage).alt) //is image alt*

-*document.CreateElement("tagName")*

-If create child **, *<i>*, etc. and append textNode to it as child, textNode will be **

Attributes
-JS DOM element objects often has same property names as HTML elements, and thus they can be accessed with . and changed, tested, with =, ==, etc.

-Custom named attributes on HTML elements must be accessed via *node.getAttribute("attributeName").* Suggested to prefix custom HTML attribute with *data-* or some other prefix, so don't accidentally conflict with existing core attr name.

-*node.setAttribute("attributeName", "value")*
    -Can also just set with *node.attributeName = "value"* for most core attributes

Layout
-*node.offsetWidth* - the width the element takes up in pixels
-*node.offsetHeight*

-*node.clientWidth* - the space inside the element, ignoring border width
-*node.clientHeight*

-*node.getBoundingClientRect()* - returns obj with *top, bottom, left, right* properties indicating pixel positions of element relevant to screen position

-Note that every time DOM changes layout or access layout via above methods, must recompute DOM. If does a lot, slow program.

Styling
-Reminder: Can pass in CSS style to HTML element by adding *style="property: value"* attribute to an HTML element. ex. *<a href="..." style="color: green">*

-Can manipulate *all* CSS styling rules and properties by calling *node.style.propertyName* on. Can then change, check, etc. with =, ==, etc..

-Note: CSS property names with hyphens in name (ex. background-color) usually have hyphen removed and use cap (backgroundColor) instead

Query Selectors
-*document.querySelectorAll("selectorString")* - returns a *NodeList* containing all selectors of exact same selector.

-Can pass any selector used in CSS (ex. "p", ".animal", "p .animal", "p > .animal", "#cat", etc). Call *array.from* on NodeList if want to interact like array (ex. call for...of on).

-Note that *NodeList* itself does not hold live elements, but can get values from, modify nodes separately

Positioning and Animating
-By giving an element a *position: relative*, can then animate by running through a function that slowly changes the position, until some end point reached, etc. Note, if try and update DOM via loop, will likely freeze page due to too many updates and slowdown.

Handling Events

Event Handling
-The browser has a variety of "events" that occur when you interact with browser nodes in some way. Ex. click on button fires "click" event and creates an *Event* object. Push of key triggers *keydown* event and creates an *Event* object. Event listeners added to nodes, and catch specified events that may fire. Event handlers are *f*unctions registered to respond to a browser event, such as a button click.

-Event listener - object that responds when a specified *Event* (ex. *MouseEvent, DragEvent*) occurs. Calls parameter specified function, which "handles" event.

- *someNode.addEventListener("eventObjectType", handlingFunction)*
- *someNode.removeEventListener("eventObjectType", handlingFunction)* - removes existing

Event Objects
-When handling function is registered to event, if handling function is set to take in an event parameter, it can gets passed an *Event* object
    -ex. *function someActions(event) {*

event.Variousproperties //properties, such as type of event
}

-ex. *event.button* //property value: 0 if left button click, 1 if middle, 2 if right

- *event.target* - property on *Event* object that refers to node it originated from.
-ex. *someEvent.tartget.nodeName == "BUTTON" //*if event occurred on button, true

-If want to create event that applies to many objects, create *let* that holds reference to a *document.querySelector("selector")* and add event listener to. Will apply to any node that matches selector.
-Ex. *let link = document.querySelector("a").addEventListener(.....);*

Propagation
-If event happens on parent node, event will also register with parents on that node that are also set to handle that event. Most specific handler acts first, least specific last. If button inside paragraph inside window, and all respond to "click," button will handle first, then paragraph, then window.

-"Propagation" from child to parent until at root

-Can stop propagation by calling *.stopPropagation()* on event (ex. inside event handler, at end of)
-Can add multiple *eventListeners* of the same type to the same *Event* type.
-ex. Add listener to print a message when mouse wheel scrolled.
Add a second listener with function that calls *stopPropagation()*

Key Events
-*keydown* event when button pressed. *keyup* when released. Specific key value held in *event.key*. Check value by performing *event.key == "w"*, etc. Can check values of modified keys by modified value (ex. shift + h becomes "H") Regexp useful for specifying alphanumeric ranges.

-*event* properties, boolean: *.shiftKey, ctrlKey, altKey, metaKey* - true if one of these keys was held as part of event. Ex. *//shiftKey == true for "H" and "E"*

Mouse Clicks
-*mousedown* and *mouseup* events for click. Occur on DOM nodes directly under pointer.

n-*click* event fires for node that contained by quick and release. Ex. click on image, move off image to paragraph. *click* fires on node that contains img and paragraph.

-*dblclick* for double click event. Note: *click* event for first click fires first.

-*clientX* and *clientY* - properties. coordinates of event in pixels, relative to top left corner of window
-*pageX* and *pageY* - properties. coordinates of event in pixels, relative to top left of whole document

-Can set location of object in page by setting *someNode.style.left* and *someNode.style.top* to desired coordinates

Mouse Motion
-*mousemove* event fired when mouse moved

-Often use this event for mouse dragging: create let for last mouse position. Add *mousedown* event to element you want to move. Check for *0* button type (right click). Add event inside *mousedown*

listener

-*event.buttons* - property. 0 = no buttons down, 1 = L button down, 2 = R down, 4 = middle down. Not to be confused with button (singular) property.

Touch Events
-A click on a touchscreen will generate *mousedown*, *mouseup,* and *click* events. No *mousemove*.

-*touchstart*, *touchend*, and *touchmove* - equivalent to *mousedown*, *mouseup*, and *mousemove*
-Since can touch in multi place on touchscreen, each event holds an *event.touches* property, which holds and array of holding, *clientX, pageY*, properties, one object per point touched

Scroll Events
-*scroll* - event fired when mouse scrolled over element
-Useful with *document.body.scrollHeight* property

-*innerHeight* - global JS binding. Can be called just with that name (*no document*. needed). Gives height of window.

Focus Events
-*focus* - event. Fired when focus given to element.
-*blur* - event. Fired when focus removed from element.
-Events do not propagate

Load Event
-*load* - fires when page finishes loading. Fires on *window* and *document.body*
-Do not propagate
-Useful for waiting to run *x* until page fully loads
-*beforeunload* - event. Fires on same as *load*. Just before page is navigated away from (ex. link click, tab closed). Useful for stopping page from closing before user saves data, etc.. If return *non-null* value from event handler, browser will show pop-up with returned value asking user if if they really want to navigate.

Event Loop
-When triggered, browser events are scheduled, then run after other scripts finish running. Because of this if running a loop in a handler, page will become slow and laggy if other handlers have to wait for loop before running.

-Web workers handle above stalling issue. Run alongside main script in own environment, no scope overlap. Runs functions and communicates output to main script.

-Create worker instance with *new Worker("filename.js"),* then access methods via *instanceName.method()*

Timers
-*setTimeout(function, ms-before-next run)* - global method
-If create set timeout function as a reference to a let, etc., then call *clearTimeout(letName)*, timer will will be removed

Debouncing
-Setting a timer for an event handler to only run *x* processing every *y* milliseconds, to handle rapid

clicking, etc.

HTTP & Forms

-Review "JS & The Browser" section of notes for basic HTTP/TCP notes

The Protocol

-Review: browser looks up ip associated with url, opens TCP connection on port 80. Sends request. Server responds. Browser shows updates.

-*GET* - A request made by the browser. Syntax: *COMMAND /resourceRequesting HTTP/version*
  -ex. *Get /page.html HTTP/1.1*

-Other common request methods: *DELETE, PUT* (create or replace), *POST*

-Server responds in for *HTTP/version statusCode string*
  -ex HTTP/1.1 200 OK
-Status code first digits: 2 = request succeeded, 4 = request error, 5 = server error

-Request first line (ex. *GET*) can be followed by any number of headers in form *name: value* to specify extra info
  -ex. *Content-Length: 65585* //size of html doc in bytes
    Last-Modified: Thu, 04 Jan 2018 14:05:30 GMT
-A couple headers, like *Host* are required

-*Put* and *Post* requests can be followed by a blank line, then a body which contains data being sent

Browsers & HTTP

-See "Sending Form Data" section in HTML notes

-Review of HTML notes: <*form*> element includes *method=""* and *action=""* attributes. *method* holds type of method (ex. *method="GET"*). *action* holds where form data is sent to (ex. *action="example/message.html"*). *GET* appends requests to end of URL (ex. *.../message.html?name=Jean&message=Yes). GET* to get. *POST* to send. Cannot simply send to text file on server though and need PHP, JS, etc. to process this. HTTP only handles requests/responses, what is being sent/requested and where to send to.

-JS will decode messages with special HTTP chars via *decodeURIComponent* global method
  -ex. *console.log(decodeURLComponent("Yes%3F"))* //logs *Yes?*

Fetch

-**Successor** of AJAX

-JS global interface for making HTTP requests. Uses promises.
-Calling *fetch* returns a promise, that resolves to a *response* obj that holds info on server's response, which can be retrieved via *then* method, which takes a function as an arg to process response

-ex. *fetch("example/data.txt)".then(response => console.log(response.status));*

-Syntax: *fetch("urlRequesting")* - If no protocol name (ex, *http:*) in *fetch*, treats url as relative (pointing to folder page is hosted on)

-*response.headers.get("key-name")* - returns map like object where keys are headers

-*fetch* has a *.text()* method, which is called on the *response* object and returns a promise that is resolved to content of response via *text*. Access via *.then(handlingFunction)* method called on *.text()*.

-ex. *fetch("example/data.txt").then(response => response.text()).then(text => console.log(text));*

-*fetch* also has a *.json()* method, that acts the same as *.text()*, but whose promise resolves to json content (or is rejected if not json). If want to convert json to string, call *JSON.stringify(someJson)* on
-Can use *json()* to fetch rest API data

-Can specify methods via *fetch* via *fetch("urlHere", {method: "DELETEetc"})*
-Can set headers via *fetch("urlHere", {headers: {Content-Length: "65585"} });* Some headers (ex. *Host*) set by browser by default.

HTTP Sandboxing
-By default, browser prevents scripts from making HTTP requests to other domains (ex. could not request data from *google.com* from local host) via cors, which wraps HTTP requests
-Can bypass this protection by including header: *"Access-Control-Allow-Origin": '*'*
-Note that site requesting from must also allow cors. It if doesn't, request will fail. If this occurs, route through cors proxy by setting url as
https://cors-anywhere.herokuapp.com/https://www.myUrlHere.com/

Appreciating HTTP
-aka, models for communicating between a client-side JavaScript program and the program on the server

-remote procedure calls - function is running server-side and browser makes request to server including function name & args. Response contains returned value.

-can also do by passing info through data, such as a json file, where information is transferred to and from server via GET, PUT, etc in json, etc., then referenced by browser/server-side functions

Security & HTTPS
-Make sure to prefer SSL and reference https when available

Form Fields
-See HTML and CSS notes for more details on forms

-Forms must be submitted as a whole, but can assess individual form values with JS to modify application, DOM, etc.

-When <*select*> menu value changes, *change* event fired

Focus
-Can set form to focus on element by calling *.focus()* on JS DOM element
-Remove focus by calling *.blur()* on element

-ex. *document.getElementById("providence").focus();*

-Can set *tabindex=#* attribute in html to change order focus occurs when pressing tab. Ex. set *tabindex=0* on element at top of form, then *tabindex=1*, on third element down, to skip second

element when tabbing. Setting *tabindex=-1* will cause element to be skipped.

Disabled Fields
-Can set field as disabled by calling *.setAttribute("disabled", "true")* on JS DOM element

Form as a Whole
-Fields contained within a <*form*> can be referenced via *.elements* property on form, which contains a map-like structure that can be accessed via name or index. Items in *elements* contain key/value pairs for html form fields

-Ex. *console.log(someForm.elements[0].type);* //prints type of first form field

console.log(someForm.elements.thePassword.type)
//will print type of form field for field that has attribute *name="thePassword"*

-Review: <*button type="submit"*> will submit form when clicked or [Enter] pressed
-By default, when submit form, browser navigates to page indicated in form*'s* html *action="urlHere"* attribute, using *GET* or *POST* request. Before navigation occurs, *submit* event fired from form. If call *.preventDefault()* on *submit* event, this will prevent navigation.

-Can use *submit* event to run form input validation or call *fetch* inside handler to submit to server without page reload

-Can create *FormData* objects by via *FormData* API by calling *new FormData(someForm).* *FormData* object holds

-Reminder that can also get current value of a form field by referencing the *.value* property of the field JS dom element

Text Fields
-<*textarea*> and <*input*> type *password* and *text* JS nodes have *selectionStart* and *selectionEnd* properties. If nothing selected, both are character cursor is at. If selected text, are numbers for start and end char of selection. Useful for replacing text.

-*change* event fires on text fields after change occurs and focus is lost. *input* event fires every time a char is entered, deleted, etc.

Checkboxes & Radio Buttons
-checkbox node has boolean *checked* property
-radio buttons fire *change* event when checked or unchecked

Select Fields
-review: <*select*> is dropdown menu allowing one selection or numerous if set with boolean *multiple* attribute via [ctrl]+click

-*value* property for <*select*> node holds value of currently selected <*option*>
-*options* property for <*select*> node holds array-like obj containing all *option* values for a *select*
-<*option*> nodes have *selected* property

File Fields
-<*file*> nodes has *files* property which is an array-like obj of all selected files chosen, as *file* also supports *multiple* attribute

-objects in *files* array have *name* (filename), *size* (bytes), *type* (ex. *image/jpeg)* and other properties

Storing Data Client-Side
-*localstorage* - global JS object. Has *setItem("key", "value")* and *getItem("key", "value")* methods for storing and retrieving strings assigned to names.
-l*ocalstorege* exists even on browser close until removed. Can remove with *remove("key")*
-*sessionStorage* - very similar global object, except clears on browser session end

Node.js

-A backend environment allowing i/o between server and client. Allows scripts to run server-side, and interact with clients, or simply perform actions server-side only, such as as a sort script for file sorting on a local disk or a CLI web scraper. Helps better streamline asynch programming to transfer data between multiple clients at the same time.

-Commands run from system terminal

The Node Command
-*node scriptToRun.js* - runs script on server. Output displayed in terminal. If run without script arg, displays prompt where you can interactively enter and run JS code. *process.exit(0)* to exit.
-*process.arg* - holds args passed to node. Array. Can call when inputting code from command line.

Modules
-Can import relative path moduleX to be used in ModuleY.js file by calling global *require(/path)*, where paths follows standard ./, .., / etc. style.
-Can also import built in or installed module (ex. React modules) via *require("moduleName")*
-Don't forget to include *export* statements if wish to import modules
-Can omit *.js* when importing

Installing with NPM
-Similar in use to linux package managers
-*npm install packageName* - fetches and installs. Installed packages in *node_modules*.
-Load via *require("ini")* or *import {something, somethingElse} from ...*

Package Files
-When run *npm init*, *package.json* file created. json file holding author name, description, dependencies, etc.

Versions
-Node generates version num in 2.3.0 semantic versioning format, where first digit increments when functionality is added that breaks compatibility. Second digit increases when functionality added that doesn't break compatibility.
-If version has ^ prior to it, any version version of the first digit may be called, as long as the second digit is greater than or equal to specified
-Can publish a package with *npm publish* in dir that holds *package.json* file. Published packages must be uniquely named and will be available to other NPM users.

File System Module
-*fs* module. Contains functions for working with files and dirs. Import with *require()* or *import*

-*readFile("filePath/name.txt", "utf8", handlingFunction)* - reads in file of specified char encoding and passes to handling function

-*writeFile("filePath/name.txt", "text to write", errorHandlingFunction)* - writes text to file in utf-8 or produces error as specified by *errorHandlingFunction* if write fails

-*readdir("filePath", callback)* - callback function gets *files* array containing list of files in path or *err* if fails

-*rename("oldPath", "newPath, callback)* - callback gets *err* if rename fails

-Node also can call functions which return promises, for asynch exe

The HTTP Module
-*http* - Creates an runs and http server, which can take and respond to http requests

-Basics:

```
let server = createServer((request, response) => {      //request = client req, resp for resp
                response.writeHead(…)                   //writes response headers
                response.write(…)                       //writes response body
                response.end()                          //end of response
                });
server.listen(8000);            //opens connections for port localhost:8000
```

-Can act as request via *request({request}, response.method);*
requestStream.end();

-Many node packages available on NPM to make http server interaction less verbose, like *node-fetch, express.js,* etc.

Streams
-Writable stream - an abstraction for writing streaming data to a destination (ex. request object returned from http *request*).
-*write* - method for node writable streams to write with. Take a string or of *Buffer* object as input to write to spefied destination. Also can take a callback to call when done writing.
-Can write stream to file with *createWriteStream* from *fs*

-Readable streams have *data* and *end* events, where *data* fires when data comes in, *end* when stream ended. Can read from via *createReadStream* from *fs*