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 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
-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 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 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 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"*) - 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)

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
-*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) => {
            return 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

-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.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 consistant, 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 defintion*
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 function.
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

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(type) { //leave as *"constructor"* as methods as *new* calls this name
this.type = type;
}
//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 thuse 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 ovveriden in instance and prototype and other instaces created from prototype unaffected.

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

Maps
-Datastructure 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 similiar 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 symbolA = "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 datastructure

-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 calles on the classe's *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 Tempreature((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 ovveride 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 controversal, because unlike encapsulation or inhertitance, 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 specificed 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 pre-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 occuring 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 vaues 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 handeling errors and return a special value (ex. *null*) or displaying an error to user. Display often prefered.

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 similiar 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 handeling 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

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

Creating a RegEx
-Can build with *RegExp()* contructor 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*

Sets of Chars
-If define pattern within [ ], defining a set. Ex. */[012345]/*. Can be quick range set with hypen: */[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 inlude 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 Subexpressions
-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 numberic *(yyyy, mm, dd, hr, min, sec, ms)*. Last four args optional. UTC.

-Can take single arg as a millisecon 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*

JS & The Browser

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

-HTTP - hypertext transfer protocol. Recieves 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
-Given access to DOM by global *document* object,which has properties refering to HTML <*html*>, <*body*>, <*head*>, <*footer*>
-Forms a tree structure where nodes are <html><main><section><h1><etc> elements and nesting. Access root via *docment.documentElement*, which refers to <*html*>

Moving Through the Tree

DOM elements have properties:
-*firstChild, lastChild, nthChild, previousSibling, nextSibling*
-*childNodes* -points to array-like object holding its children. Has a .length property. Need full *for* loop with iterator to traverse.
-*children* - points to array-like object only element children, not text nodes
-*nodeType* - ex. *ELEMENT_NODE*

-Moving through DOM often smooth when done with recursion

Finding Elements
-DOM created also of white space not seen in HTML doc tags