**NodeJS Install**  
  
-Before using JavaScript (and React), we need NodeJS, which is a JS runtime environment.

<https://nodejs.org/en/download/>

**-Make sure to check “Automatically install necessary tools”.**

-Otherwise, ”Next” your way through everything.

-A command prompt will show up, just do what it says. It’ll install a bunch of junk.

**-...F o r a l o n g t i m e …**

-If your antivirus software stops it, it should be fine to allow the download to proceed.

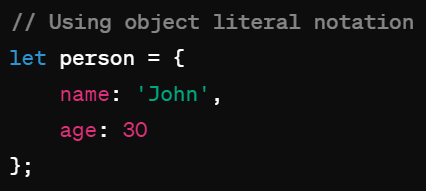
-In git bash, enter: node -v to make sure it’s installed. It’ll tell you the version you have.

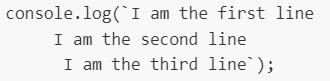
-Node Package Manager (npm) has been installed too by the way!! Try npm -v to prove it. This shows the version of npm. We’ll use this later.

**What is JavaScript? (we started these notes at 10:45 EST)**

* Not Java. Not at all.
* JavaScript is a high-level scripting/programming language generally used for **front-end development**. It has built-in support for all modern web browsers.
  + **JavaScript is how we add functionality to our webpages.**
  + With the creation of NodeJS (a JS runtime environment) it is possible to use JS as a server-side language, and plenty of applications do so.
* JS is **multi-paradigm** - it can do OOP, procedural, and functional programming etc.
  + JS is **loosely typed.** (a variable’s datatypes don’t need to be defined).
    - a = 56; //this is valid. JS assumes this is a number.
  + Official language specifications for JavaScript are called **ECMAScript.** This means that when we talk about the versioning of JS we refer to “ES#”.
    - e.g. ES6, ES7.
  + In HTML, **you will link to an external JS file using the <script> tag.** 
    - Script tags can go either in the head or the body, but the best practice is generally to **put them at the very bottom of the body.** Errors like are more likely to pop up if you don’t. Also good for Search Engine Optimization (SEO), because the skeleton/content of your webpage should show up before the JS logic gets compiled.
  + JS was originally written by a single person in ~10 days. This is probably why there are some quirks with the language that persist to this day.
    - See notes below on type coercion for an example.
  + With NodeJS installed, we can run our JS files by running “node appName.js” from a command line inside the directory holding the JS file in question.
  + 

**JS Syntax Rundown**

* Comments in JS are the same as in java: //this is a comment
* JavaScript has **variables** and **literals**, which are the fixed, hardcoded values of those variables. They can be numbers, strings, booleans, objects etc.
  +  x is the variable, and 10 is the int literal
  +  y is the variable, ”Hello World” is the string literal.
    - NOTE – String in JS can be “double quoted” or ‘single quoted’.
* JavaScript objects organize data in key value pairs as seen here:
  +  (Remember JSON? JS Object Notation)
* JavaScript has keywords like other languages that are reserved because they have specific meaning. E.g. var, let, break, case, catch, continue, do, else, if, new, throw, etc.
* JavaScript has a mechanism known as “template literals” (not to be confused with literals mentioned above). They allow us to create **multi-line Strings** with the use of `backticks`



* Semicolons aren’t necessary but I still like to use them;

\*Start HelloJS here if NodeJS installs are done\*

**JavaScript Variables**

* Variables store data values. The “=” is used for the assignment operator
  + E.g. var a = “I am a variable”
  + Identifiers (names of the variables) must start with either a letter, underscore, or $. They cannot start with numbers, though they can contain them. No spaces either.
* There are **3 ways to declare a variable in JS.** (let and const were added in ES6)
  + **var** - older, default declaration
  + **let** - newer and preferred declaration for mutable variables. let can be block scoped (see below).
  + **const** - has the functionality as let, but once assigned it can not be changed. (Like a “final” keyword in Java).
* There are **7 major data types** of JS variables:
  + **number** - includes integers and floating point numbers.
  + **string** - collection of characters surrounded by “double” or ‘single’ quotes.
  + **boolean** - true or false
  + **null** - has no value
  + **undefined** - the variable has not been assigned anything. It is only declared.
  + **object** - objects in JS are collections of key/value pairs. Uses {}.
  + **Symbol** - added in ES6, not commonly used.
* **Other data types**:
  + **BigInt** is used to hold large integers outside the ability of the standard number variable.
  + **NaN** (not a number) is technically of the data type number. It is returned when a mathematical operation encounters a problem and can not return a valid number value.
    - Like if we try to divide String “hello” by number 5
* Object subtypes:
  + **Arrays** (Big differences from Java!) - Arrays are objects that have indices, like in Java. However, **a single array can hold any number of different data types and their length is not fixed.**
  + **Functions** are also objects but they have greatly expanded functionality.
* The “typeof” operator returns the data type of its operand. Very useful for debugs.

**JS Operators**

* Arithmetic operators: +, -, \*, /, %, ++, --
* Comparison: >, <, >=, <=, !=, ==, ===
  + **=== will compare types AND value while == simply compares value** 
    - **(ie 6 == ‘6’ returns true but not 6 === ‘6’)**
  + **Eg false == 0 returns true, while false === 0 returns false;**
* Logical: &&, ||, !
* Assignment: =, +=, -=, \*=, /=, %=
* Ternary: <condition> ? <value1>:<value2>
  + This is like a one line if/else statement. If the condition is true, value1 is the result, if false value2 is.

**Control Flow**

* If/else, if/else-if/else
* for loops
  + There are 2 special types of for loops.
    - **for-in**: iterates over an object’s keys
      * let person = {name:”Bob”,
      * age:25}; // this is a JS object
      * for(let key **in** person) {console.log(person[key]);}
        + //prints Bob and 25
    - **for-of**: iterates over the values of an array.
      * let arr = [12, “Susie”, true, 0];
      * for (let value **of** arr) {console.log(value);}
        + //prints 12 then Susie then true then 0.
    - Think for-in = objects, for-of = arrays.
* while loops/do-while loops
* switch statement

**Type Coercion**

* Type coercion is **the process of converting a value from one data type to another.** There is explicit and implicit type coercion in JavaScript.
* **Explicit type coercion** is when we specify that we want a type changed to another type.
  + E.g. number(“3”), String(123)
  + //we’re simply specifying explicitly what data type we want a certain value to change to.
  + NOTE – explicit type coercion is just casting like we’ve seen in Java.
* **Implicit type coercion** is where JS will attempt to carry out your instructions by changing the type variable on its own. This happens when you apply operators to values of different types.
  + E.g. “3” \* 2; //number - JS chooses for us. Likely because you wouldn’t multiply a String
  + E.g. “3” + 2; //string - Likely because you commonly concatenate strings
* Type coercion is one of those areas where JS quirkiness really shines through

**Truthy and Falsy Values (actual technical terms)**

* In JS, **any expression or value can be evaluated as a boolean.** We use the terms truthy and falsy to indicate what boolean something will be evaluated as even if they aren’t inherently booleans.
* There are **6 values/expressions that return as false**
  + The boolean false
  + An empty string (e.g. var a = “”; //a is falsy)
  + Undefined values (variables that haven’t been given a value yet)
  + Null values
  + NaN value
  + 0
* Any other value besides these 6 will return a value of true

\*Finished HelloJS\*

**JS Functions**

* A function is a reusable group of code that can be called anywhere in the program. **Analogous to Java methods.**
* You declare a function using the “function” keyword. A function can take any number of parameters you choose.
* Functions can return values using the “return” keyword. **You do not have to specify a return type** like in Java.
* JavaScript functions are objects and can be stored in variables.
* There are various types of functions in JS:
* **Anonymous functions** have no identifier (name). Often held in a variable.
* **Arrow functions** are “one time use” functions that are written inline
  + Analogous to lambdas in Java, they are called arrow functions because the syntax uses “=>” to make an arrow
    - E.g. (var1) => {console.log(var1);}
* **Callback functions** are functions that get passed into another function as a parameter, and then the original function executes the parameterized function.
  + This is helpful in writing **asynchronous JS code.** Code generally runs from top to bottom of the file - **if code is not happening in sequence, it is considered asynchronous.**
  + Since all functions in JS are objects, they can be passed in as parameters of other functions.
* **Closures** - This is an old way of achieving encapsulation in JS.
  + This is a nested function that can access the variables and arguments of its outer function, but can no longer change them.
    - We won’t ever find ourselves writing our own, but they’re baked into a lot of fundamental JS constructs.

\*Started HelloJSFuncsScopes\*

**JS Scopes**

* JS has two major divisions of scopes: Global and Local
  + **Global scoped** variables are **accessible anywhere** in the application
  + **Local scoped** variables are **accessible only in their location**. Local locations are defined by {curly braces}
* There are two flavors of local scopes
  + **Functional Scope:** Variables declared with any keyword inside a function. They’re only accessible inside that function.
    - Within functional scope, we also have the concept of “**lexical scope**”, which is the ability of a nested function to access values defined in the parent function it was defined in. Closures, for example.
  + **Block Scope:** Variables declared inside curly braces (but not specifically a function) that are declared with the let or const keywords. They will not be accessible outside that block of code.
    - Variables declared with var are hoisted (see below) and therefore cannot be restricted to block scope. They are GLOBALLY scoped.

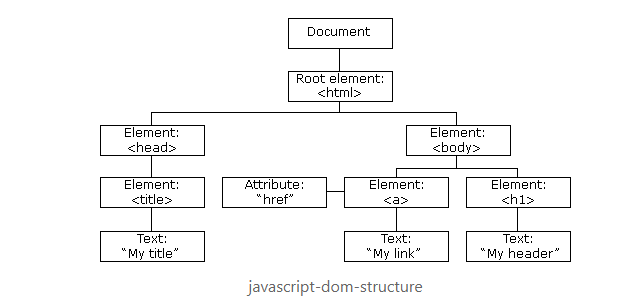
**Hoisting**

* Hoisting is a default JS mechanism where **variables (specifically vars) and function declarations are moved to the top of their scope** before code execution.
  + NOTE: The variable declarations are hoisted, but the assignment isn’t. Therefore, all vars = undefined until the assignment happens in the script.
* Variables declared with **let/const are not globally hoisted**, only to the top of the block they’re declared in.
* What’s the point of globally hoisting var but not let?
  + Global hoisting lets us do stuff like call functions before they appear in the code, but having an “undefined” value can cause unexpected behavior.
    - The addition of **“let” helped developers write more predictable code.** Trying to access a let outside of its scope leads to a ReferenceError, which can be error handled.

\*Finished HelloJSFuncsScopes\*

**Document Object Model (DOM)**

* When you attach JS to HTML via the <script> tag, it will read that document and **convert the HTML elements into a JS object that can then be manipulated.**
  + This is a good look into how JS and HTML interact under the hood.
  + JS converts HTML elements into objects, hence Document Object Model
* The DOM is created as a “tree” where the root element <html> is the root of the tree, and the other elements “branch” out from the root. **Each element in the HTML is an object in the DOM.**



* In JS, you **access the DOM with the “document” object.**
  + **DOM Selection** is the act of **accessing the elements** of the HTML file through the DOM object.
    - document.getElementById(“idName”) will return a single element with the specified ID.
    - document.getElementsByClassName(“className”) returns an Array of elements with the specified class.
    - And more!!!!!!
  + **DOM Manipulation** is JS is the act of **changing** the elements of the DOM during run time.
    - **.setAttribute()** will change a DOM element’s attribute directly
    - **.appendChild()** creates an element as the direct child of another

\*Start HelloDOM\*

**JS Events**

* Events occur **when user interaction takes place on the web page**, such as clicking a button, hovering over something, or pressing a key on the keyboard.
* When events occur, we can use an **event handler** (AKA event listener) to detect them and perform a specific action.
* Commonly used events:
  + **onclick** - when a user clicks an element, often a button.
    - You’ll be using this 90% of the time
  + ondblclick - when a user double clicks an element
  + onmouseover - when the user moves the mouse pointer over the element
  + onload - happens when the browser finishes loading the page
  + onunload - happens when the page is closed
  + onresize - happens when the window is minimized or maximized
  + onkeydown - when the user pressed down on a key.
  + onkeyup - when the user releases a key.
  + onsubmit - when a form is submitted.
* **EventListener** is what we use to detect events and perform some action
  + The **addEventListener function** is built into JS to handle various events.
    - It **“listens”** for a certain event to happen, then executes some logic.
  + The syntax: element.addEventListener(event, function, useCapture)
    - **event -** type of event being listened for (see above for some options)
    - **function -** what code to run when the event happens
    - **useCapture -** optional boolean value used to determine if the event handler should use bubbling or capturing event propagation. Read directly below on this.
      * In short, it defines how to handle multiple listeners triggering with the same event. The default is false, bubbling.

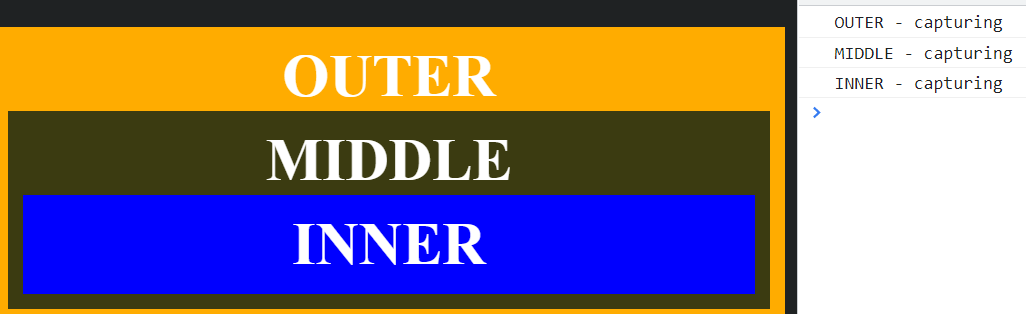
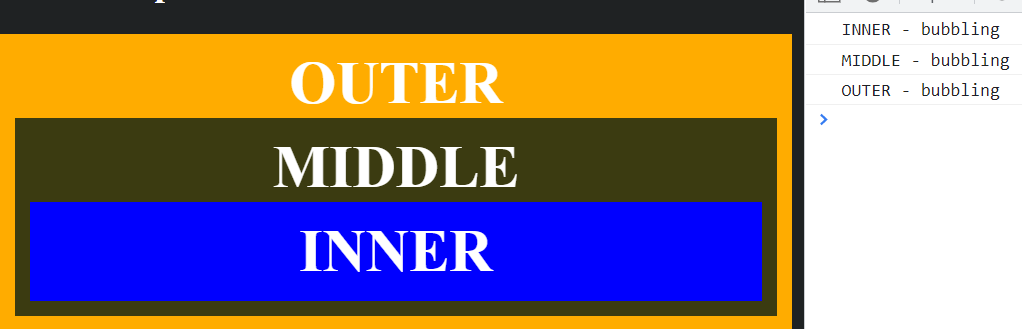
 Applying an EventListener to a button

**NOTE – we can use the shorthand to apply an event listener like this:**

 Directly calling .onclick on the button. Easier! Cleaner!

\*Finished HelloDOM\*

**Bubbling and Capturing (Event propagation) (Moving to next week)**

* Event Propagation is the “traveling” of an event through all the DOM elements that it could be related to. (All the nested elements from where the event happened.)
  + **This is where useCapture in EventListeners (mentioned above) comes in**
* We set whether we want bubbling/capturing in the eventListener using the useCapture boolean.
* **Capturing**(useCapture = true): This sets our event listener to respond the first time the event “travels” through the DOM, so **outer elements will trigger their event listener before inner elements** all the way down to the “trigger” element. (The button that was clicked for instance.)
  + Think - the outermost elements capture the innermost elements. **Capturing goes down through the nested elements.**
  + ****
* **Bubbling**(useCapture = false - default): The event then travels back up the DOM through all the elements again. This makes it so that **innermost elements will trigger before the outer elements.**
  + Think - bubbling up through the DOM elements. Bubbling goes up from the nested elements to its parent elements.
  + ****
* We can also stop the propagation process with the event.stopPropagation() method.
* NOTE: There is almost NEVER a reason to use capturing instead of bubbling. We rarely need an event to trigger in the parent first. We almost always leave the default.

**JS Fetch API (our first true look into client-server communication)**

* With JS, we can use the Fetch API which is a modern and versatile means of **sending asynchronous HTTP requests.**
* We use the **fetch() method** to **return a promise object** (see below).
  + fetch() takes two parameters:
    - A **URL** that the request is sent to
    - A **configuration** **object** which can contain multiple options that define the request.
      * This is where you can set the HTTP verb, request body, and more.
* A **promise object** represents a value that **may not be** **available yet but will be resolved in the future.** Instead of receiving the value, you get a “promise” that it’ll come later.
* Workflow:
  + The browser **sends a request to the server** and creates a **promise object for the response**
  + If the HTTP request **fails** (the response returns an error code), the promise resolves and the **Fetch API rejects the promise object.**
  + If we get a successful response, the promise resolves, and **returns response data in the Response Body.**
* Methods to access response body:
  + **response.json() - parses the response as JSON and returns a JS object.**
  + **response.text()** - returns the response as plain text.
  + There are more, we won’t use them but feel free to research
* .then(), .catch(), .finally():
  + These are methods used after a fetch request to **manage the workflow.** We can do things like format and use the incoming data, catch any errors, and perform final actions after the data is consumed. Works similarly to a try/catch/finally in Java!
    - .then() is typically used to parse and use the incoming data
    - .catch() catches errors, provides error handling code
    - .finally() is used for anything you want to happen after the previous actions complete.
* Keywords
  + **async** - this is added to functions to tell them to return a promise, rather than directly return a value
  + **await** - in an asynchronous function, await will pause the function until the promise is returned
  + \*\*These two keywords will go hand in hand
* These two keywords help us accomplish **asynchronicity**. They create functions that are asynchronous – the function will not interrupt the main functionality of the app. The app will continue working while the function works in the background.

\*HelloFetch\*

[will post our fetch request here]

**“this” Keyword**

* In JavaScript the “this” keyword has multiple meanings based on where it’s used.
  + **“this” alone** refers to the global Object. (The window object)
    - Probably won’t worry about this in our projects
  + **“this” in event handlers** refers to the HTML element that receives the event.
    - E.g. <button onclick = this.style.color=”green”>click me</button>
    - As with other inline styling, this isn’t best practice!!!
  + **“this” in Object Method Binding** (like a constructor) refers to the object.
    - E.g. var person = function (name, age) {

this.name = name;

this.age = age;

}

**NOT ON QC BUT POTENTIALLY USEFUL----------**

**Timing Events**

* Timing events are used to **automate or run tasks after certain intervals of time.** The two main timing events we see are setTimeout() and setInterval()
* **setTimeout** takes a certain function, and an amount of milliseconds to wait before executing that function.
  + E.g. setTimeout(myFunction, 2000)
    - myFunction will run 2 seconds after being called
* **setInterval** takes a certain function, and executes it multiple times after a certain interval.
  + E.g. setInterval(myFunction, 2000)
    - myFunction will run every 2 seconds
* **Big picture:** setTimeout() invokes a function once after a wait time. setInterval() invokes a function repeatedly with a wait time between executions.
  + There is a nice little example in the curriculum notes if you want to look into these further

**Strict Mode**

* Declaring “use strict” in JS means that you will not be allowed to use undefined variables, any keywords as variable names, and restricts some other niche instances. **Basically, it removes the quirkiness of JS.**
* It was implemented in ES5
* It can be declared at the global level (just as the first line of your script) or functionally scoped (first line of your function.)

**JS Prototypal Inheritance**

* All JS objects have a **prototype**. This is implemented through the **\_\_proto\_\_ property.** This property is used to define **inheritance in JavaScript.**
  + An object’s \_\_proto\_\_ can be set to reference another object. This will make that referenced object the parents of the object that references it.
* The **top prototype** of all objects in **Object.prototype**. This is the value that is assigned to \_\_proto\_\_ by default.
  + Analogous to the Object class in Java. The parent of all classes.

**JS Classes**

* **Classes are a special type of function in JavaScript**
* You can define a class with a **class declaration**
  + You use the “class” keyword with the name of the class (like declaring a function but without parenthesis for parameters).
* You can also define a class with a **class expression**
  + You define a variable and then assign it to an object with the class keyword
* Classes can have a special function in the class called a constructor, which functions like a constructor in Java (initializes the object by assigning values to its variables.)