Meta Elevate - Meta Front End Developer

1. DOM
    a. Document Object Model
    b. Tree structure of the objects
    c. The DOM has a series of objects, each representing a single HTML element
    d. At the root of the DOM is the HTML object
    e. HTML object contains the head and body objects
    f. Head object houses the title object which holds the text object
    g. The body object houses the DIV objects which then can contain header, paragraph, and text objects such as ‘<div>’, ‘<h1>’, ‘<p>’, and ‘text’.
    h. The Document Object Model allows you to update all HTML elements on a web page.
    i. Many JavaScript libraries and frameworks rely on the DOM, one of these libraries is the react library.
2. Webpages
    a. Typically have hundreds of elements
3. Common DOM and Javascript uses:
    a. Updating the elements on an HTML page based on interaction from the user
        i. Notification for incorrect password
        ii. Notification for message received
        iii. Interaction with a video player
4. Accessibility
    a. In this course we will learn the proper way to use the DOM and HTML elements so that the content is accessible to everyone
    b. Having text that is not contained within proper tags like paragraphs or heading tags makes it harder for assistive technologies to interact with the content.
5. Extra resources
    a. HTML Elements Reference (Mozilla)
        i. https://developer.mozilla.org/en-US/docs/Web/HTML/Element
    b. The Form Element (Mozilla)
        i. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/form
    c. What is the Document Object Model? (W3C)
        i. https://www.w3.org/TR/WD-DOM/introduction.html
    d. ARIA in HTML (W3C via Github)
        i. https://w3c.github.io/html-aria/
    e. ARIA Authoring Practices (W3C)
        i. https://www.w3.org/TR/wai-aria-practices-1.2/
6. Selecting and Styling
    a. The HTML is the frame and stricture of the building
    b. CSS is the paint, wallpaper, fixtures, artwork, and overall style or look and feel. In other words, CSS tells the web browser how to display HTML elements on the screen.
    c. CSS rule

i. A declaration block starts with a left curly brace and ends with a right matching curly bracket
ii. In between the curly brackets are the style declarations
iii. The first element of the CSS rule is he selector
    1. This indicates the CSS element or elements we want to style.
iv. Example
    1. H1 {
       Color: green;
       }
        a. H1 is the selector
        b. Color is the property
        c. Green is the value
v. ID selector
vi. “What if I only want to style one particular type of element on a web page; ie. not all h1 elements but just one h1 element - id selector
    1. In HTML document, Add an ‘id’ to the html tag - <h1 id=”header1”>
    2. In CSS document, reference id with # like #header1
        a. Example
            i. #header1 {
               Color: green
               }
vii. Class selector
    1. Good for selecting multiple elements of the same class without selecting all elements
    2. Tighter scope of element selection than root element.
    3. Looser scope of element selector than ID.
    4. Hierarchy should be
        a. Root element
        b. Class(es) of similar elements
        c. IDs for very specific elements
    5. IDs and classes can be used interchangeably I believe but should not be
    6. Element with class selector:
        a. HTML - <p class=”introduction”>
        b. CSS:
            i. P.introduction {
               Margin:2px;
               }
    7. Descendant selectors
        a. Descendant selectors are useful if you need to select HTML elements that are contained within another selector.
            i. Example:

                    1. HTML:
                       <div id="blog">
                       <h1>Latest News</h1>
                       <div>
                    2. CSS:
                       #blog h1 {
                           color: blue;
                       }
                b. The CSS rule will select all h1 elements that are contained within the element with the ID blog. The CSS rule will not apply to the h1 element containing the text Archives.
            8. Child Selectors

7. Text and Color
    a. RGB
        i. A color model that creates colors by adding varying degrees of red, green, and blue together
        ii. This is how the human eye sees color
    b. RGBA
        i. An extension of the RGB color model which adds the alpha channel. The alpha channel adds opacity or transparency of the color
    c. HSL
        i. A newer color model that is defined as hue, saturation, and lightness.
8. Box Model
    a. Hiearchy
        i. Margin
        ii. Border
        iii. Padding
        iv. Content
    b. Document Flow
        i. Block vs inline elements
            1. Block take up the full width of the screen and the height of the content
            2. Inline take up as much width and height of the content and can be on the same line, hence the name inline
    c. Additional CSS resources
        i. CSS Reference (Mozilla)
        ii. https://developer.mozilla.org/en-US/docs/Web/CSS/Reference
        iii. HTML and CSS: Design and build websites by Jon Duckett
        iv. https://www.amazon.com/HTML-CSS-Design-Build-Websites/dp/1118008189/
        v. CSS Definitive Guide by Eric Meyer
        vi. https://www.amazon.com/CSS-Definitive-Guide-Visual-Presentation/dp/1449393195/
9. Working with libraries

a. Libraries and frameworks are called dependencies
    i. Components can have multiple dependencies. A dependency can have a dependency of its own. This is called a dependency tree. Large projects can have 100s of dependencies in its tree
    ii. Dependency relying on other frameworks and libraries = dependency
    iii. Package manager takes care of downloading all needed dependencies
    iv. NPM is the most common package manager for Front End Development
    v. Node Package Manager = NPM
    vi. You will use a bundling tool to gather all your dependencies and combine them so that they can be referenced from your HTML file.
b. A web page and its components need the dependencies to be uploaded/present on the web server in order to render on the internet

10. Introduction to responsive design
    a. 3 principles
        i. Flexibile Grids
        ii. Fluid Images
        iii. Media Queries
11. Getting started with bootstrap
    a. <div class=”container”>
        i. <div class =”row”>
            1. <div class =”col”>
                a. <h1>Our Menu</h1>
                b. <h2>FalafelM/h2>
                c. <p>Chichpea, herbs and spices.</p>
                d. <img src”image.jpg” class=”img-fluid”/>
                    i. The class “img-fluid” ensures that the image scales to its parent column’s width
    b. The first element that needs to be added when setting up a bootstrap website is the container element.
12. Using Bootstrap styles
    a. Bootstrap has a large CSS library built by Bootstrap’s developers using thousands of use cases
    b. Class Fixes
        i. Response breakpoints
            1. Extra small < 576
            2. Small (sm) >= 576px
            3. Medium (md) >= 768px
            4. Large (lg) >= 992px
            5. Extra large (xl) >= 1200px
            6. Extra extra large (xxl) >= 1400px
        ii. A class abbreviation does not exist for Extra small because this is the default breakpoint in Bootstrap CSS rules
    c. Modifiers

            i. Bootstrap modifiers add a CSS class to change the visual style of components.
            ii. (in context of alerts)
                1. Primary <- uses bootstrap's default color; blue
                2. Secondary
                3. Success
                4. Info
                5. Warning
                6. Danger <- uses color red
                7. Light
                8. Dark
    d. <div class="col-6"> becomes <div class="col-lg-6"> (large)

13. Bootstrap Grid
    a. Building a website using responsive design requires a responsive grid and responsive breakpoints
    b. Bootstrap provides both of these as part of its library
    c. The bootstrap grid system helps us to create web page layouts through a series of rows and columns that house our content. The bootstrap grid system always has a container, rows, and columns
    d. The container is the root element of your grid.
14. Bootstrap components
    a. Bootstrap includes a pre-made set of UI elements and styles to help you build your website.
        i. These are called bootstrap components
    b. Card component
    c. Alert component
    d. Alert-info component (makes alert blue)
15. SPA = Single Page Applications
    a. Examples
        i. Social Media Network
        ii. Maps app
        iii. Messenger App
    b. Traditional multi-page websites became server intensive as it takes more resources to load individual pages.
    c. Two ways of delivering code to browser:
        i. All at once - Bundles
        ii. Dynamically as needed - Lazy Loading
    d. User interacting with the page updates the page contents
        i. PUT request to server
            1. Server returns JSON - which is less resource heavy than returning a full page
16. What is React?
    a. Available since 2013
    b. Open source Javascript library

   c. Core contributors and companies maintain it
   d. Can develop single page applications
   e. Can also develop mobile applications with React Native
   f. The key concept behind React is that it allows you to define components that you can combine to build a web application.

17. React component
    a. Small piece of user interface
       i. Examples
          1. Music player
          2. Photo gallery
    b. This component model allows:
       i. Isolated Development
       ii. Isolated Testing
       iii. Re-using Components
18. How React Works
    a. A React Element has a 1:1 relation to the HTML Element that displays on the page.
    b. React uses the virtual DOM to update the actual/Browser DOM when it needs to.
       i. This ensures your application is fast and responsive to users
       ii. Reach checks for changes to the virtual DOM and pushes updates from the virtual DOM to the Brwoser DOM as needed. This means elements of the DOM that are not updated remain the same - reconciliation
       iii. React updates the virtual DOM and compares it to the previous version of the virtual DOM. If a change has occurred, only that element is updated in the browser DOM. Changes on the browser DOM cause the displayed webpage to change.
       iv. React builds a representation of the browser Document Object Model or DOM in memory called the virtual DOM. As components are updated, React checks to see if the component's HTML code in the virtual DOM matches the browser DOM. If a change is required, the browser DOM is updated. If nothing has changed, then no update is performed.
       v. As you know, this is called the reconciliation process
       vi.
    c. React Fibre Architecture
       i. The principle of the React Fibre Architecture is its like a priority system
          1. What is most important to load and when?
    d. Component Hierarchy
       i. Every React application has at least 1 component
          1. Begins with root
             a. Example
                i. App
                   1. NewItemBar
                   2. ShoppingList
                      a. ShoppingItem

        b. ShoppingItem
        c. ShoppingItem
        d. ShoppingItem

19. Programming With Javascript
    a. Week 1
        i. High-level and low-level languages
            1. The zeros and ones of binary code is a low-level language because it's closer to being understood by a CPU. JavaScript on the other hand is a high-level language. This means that it has to be converted to binary code so that a CPU can work with it.
            2. As a high-level language, JavaScript does need to be converted to binary code so that a CPU can work with it.
        ii. JavaScript invented in 1995?
        iii. Jquery was the first library to solve a lot of browser compatibility issues
        iv. React came along in 2011 and then some others followed
        v. Other Frameworks and Libraries:
            1. Vue.JS
            2. Angular
            3. Knockout
            4. Backbone
            5. Ember
        vi. With millions of websites containing JavaScript code from different versions and libraries, there is a lot of old code. This is known as a legacy code. While you probably won't use jQuery to build a modern website today, you might still come across it in a project that is still actively running. But don't worry, sometimes beginners think they have to learn or even master all the different technologies associated with JavaScript. However, that's not really necessary. To be a well-rounded developer, you need to learn and master the basics of plain JavaScript without the frameworks. Once you have this foundation, the pathway will become easier for you to learn a framework built on top of JavaScript, such as React.
        vii. Variables
            1. Declare by using
                a. Var
                b. Let
                c. Const
        viii. Data Types
            1. 7 primitive Data Types in JS:
                a. String - text in single or double quotes
                b. Number - numerical values
                c. Boolean - true/false
                d. Null - absence of value
                e. Undefined - unassigned value

f. BigInt - like an extra large box that can hold a very large range of numbers
g. Symbol - Unique identifier. lke having a bunch of boxes that all say “Dishes” but each with different serial #s

ix. Operators
1. Are used to perform operations on variables and values.
2. An operator is used to manipulate individual data items and return a result
3. Assignment Operator - ‘=’
4. Arithmetic Operators
    a. ‘+’ - addition
    b. ‘-’ - subtraction
    c. ‘/’ - division
    d. ‘*’ - multiplication
5. Comparison Operators
    a. ‘>’ - greater than
    b. ‘<’ - less than
    c. ‘==’ - equal to (compares value not type)
    d. ‘===’ - equal to (compares value and type)
    e. ‘!=’ - not equal to
6. Logical operators
    a. ‘&&’ - Checks for both conditions to be true
    b. ‘||’ - checks for at least one condition to be true
    c. ‘!’ - returns false if the result is true
7. Modulus Operator - ‘%’
    a. This checks how many times you can fit one number into the other and then return the remainder. If I type 9 modulus 8, the value of one is returned, or 16 modulus 8 returns a zero. This is because the number eight divides into the number 16 evenly. There is no remainder as represented here by the zero.
8. Additional operators
    a. Logical AND operator: &&
    b. Logical OR operator: ||
    c. Logical NOT operator: !
    d. The modulus operator: %
    e. The equality operator: ==
    f. The strict equality operator: ===
    g. The inequality operator: !=
    h. The strict inequality operator: !==
    i. The addition assignment operator: +=
    j. The concatenation assignment operator: += (it's the same as the previous one - more on that later)

x. Numbers

                1. The number of data type is a foundational part of JavaScript as a programming language because it represents both integer and decimal point numbers.
                2. Without it, you wouldn't be able to code.
                3. Foundational data type that represents integers and decimal points
        xi. Strings
                1. Strings are used to represent and work with a sequence of characters while programming in JavaScript.
                2. You can use single or double quotation marks to make a string literal.
        xii. Booleans
                1. Used to check if a statement is true or false
                2. Booleans only have two values: true and false
                3.
    b. Week 3
        i. Programming Paradigms
                1. Human languages have many variations and forms such as formal/informal and slang. They all perform the exact same function, communication.
                2. Programming languages are similar and have many styles.
                        a. Just like human languages no one style is better suited than the other.
                3. Two commonly used paradigms:
                        a. Functional Programming (FP)
                        b. Object-Oriented Programming (OOP)
        ii. camelCase
        iii. Functional Programming
                1. In functional programming, data and functions that operate on it are clearly separated, not combined inside objects.
                2. JavaScript allowing me to use the return keyword as described above, I can have multiple function calls, returning data and manipulating values, based on whatever coding challenge I have in front of me.
                3. Being able to return custom values is one of the foundations that makes functional programming possible.
                4. In functional programming, we use a lot of functions and variables.
                5. When writing FP code, we keep data and functionality separate and pass data into functions only when we want something computed.
                6. In functional programming, functions return new values and then use those values somewhere else in the code.
        iv. Object-Oriented Programming

            1. In this style, we group data and functionality as properties and methods inside objects.
            2. OOP helps us model real-life objects.
            3. It works best when the grouping of properties and data in an object makes logical sense; meaning the properties and methods “belong together”.
        v. “To summarize this point, we can say that the Functional Programming paradigm works by keeping the data and functionality separate. It's counterpart, OOP, works by keeping the data and functionality grouped in meaningful objects.”
        vi. There are many more concepts and ideas in functional programming
            1. Here are some of the most important ones:
                a. First-class functions
                    i. It is often said that functions in JavaScript are “first-class citizens” which means that a function in JavaScript is just another value that we can:
                        1. Pass to other functions
                        2. Save in a variable
                        3. Return from other functions
                    ii. A function in JavaScript is just a value; almost no different than a string or a number.
                        1. For example, in JavaScipt it is perfectly normal to pass a function invocation to another function.
                b. High-order function
                    i. A high-order function is a function that has either one or both of the following characteristics:
                        1. It accepts other functions as arguments
                        2. It returns functions when invoked
                        3. There is no “special way” of defining high-order functions in JavaScript. It is simply a feature of the language. The language itself allows me to pass a function to another function, or return a function from another function.
                c. Pure functions and side-effects
                    i. A pure function returns the exact same result as long as it’s given the same values.
                    ii. Another rule for a function to be considered pure is that it should not have side-effects.
                    iii. A side-effect is any instance where a function makes a change outside of itself
                        1. This includes:

                a. Changing variable values outside of the function itself, or even relying on outside variables
                b. Calling a Browser API (even the console itself)
                c. Calling Math.random() - since the value cannot be reliably repeated

vii. Function Calling and Recursion
    1. Functions that repeat tasks are helpful; unless they run endlessly
    2. What recursive functions are
        a. When a function calls itself, this is what is known as recursion
    3. How to avoid getting stuck in an infinite loop
        a. Need a condition to stop. In the provided example, a variable was added above the function called counter and the condition ‘if counter = 0: return;’ within the function. After that middle line the function called itself again
            i. The order is important

viii. Scope
    1. All about accessibility
        a. It determines which parts of the code are accessible and which parts are inaccessible.
        b. For example what variables can a function access within code?
    2. Scope Chain in Javascript
        a. The code that exists outside of a function is referred to as global scope
        b. All the code inside of a function is known as local scope or function scope.
        c. If a variable is defined within a function, then you can say it is scoped to that function. This is also known as local scope.
        d. This chain of scope references is referred to as the Scope Chain.
        e. Each function keeps a reference to its parent scope
    3. A nice way to think about how scope works in Javascript is a two-way mirror
        a. This is a piece of glass where only one side is transparent
        b. For example, if a restaurant uses two-way glass, people outside the restaurant can’t see what’s happening inside but the people inside can see what is happening outside.

c. Additional resources
    i. Here is a list of resources that may be helpful as you continue your learning journey.

        ii. These resources provide some more in-depth information on the topics covered in this module.
        iii. Mozilla Developer Network Expressions and Operators
        iv. Mozilla Developer Network Operator Precedence and Associativity
        v. JavaScript Primitive Values
        vi. ECMA262 Specification
        vii. jQuery Official Website
        viii. React Official Website
        ix. StackOverflow Developer Survey 2021 Most Popular Technologies
        x. Emojis

20. Writing Statements
    a. Conditional Statements
        i. If statement
            1. The if statement checks a condition and will execute a code block if the condition is met or true.
        ii. If Else statement
            1. Same as above with added functionality. If something is true execute nested code block under if. If not true (else) execute nested code block under else.
            2. Better suited for
        iii. Else-If statement
            1. Same as above with added functionality. If something is true execute nested code block under if. If the first statement is not true and the second or later (else if) condition is true, execute the nested code block under else if. If not true (else) execute nested code block under else.
        iv. Else statement
            1. Execute if all other conditional statements above = false
        v. Switch statement
            1. Uses case and checks for parameter as a condition on each case. Default is the last option and it is similar to else.
            2. if there are a lot of possible outcomes, it is best practice to use a switch statement because it is easier less verbose. Being easier to read, it is easier to follow the logic, and thus reduce cognitive load of reading multiple conditions.

21. Looping Constructs
    a. Loops are used to continually execute repeated blocks of code until a certain condition is met/satisfied
    b. Loops are similar to conditionals; a condition must be satisfied in order for the code to execute.
    c. Different kind of loops
        i. For
            1. Set the value of the counter
            2. Specify the condition - i < 3;

        3. Increment the counter - i++
        4. For (var i = 1; i <= 3; i++) {
           ```
           For (var i = 1; i <= 3; i++) {
                   console.log(i)
           }
           ```
        5. Exit condition - works with the incrementor to prevent the loop from running forever by specifying at which value to terminate the loop.

    ii. While
        1. Similar to the for loop; however the start counter is set outside of the while loop and the incrementing is done inside the loop’s body.
        2. Code that repeats as long as a specified condition is true
        3.
           ```
           var i = 1;
           while (i < 6) {
              console.log(i)
              i = i +1;
           }
           console.log('Countdown finished!');
           ```
        4.
    iii. Nested
        1. Putting a loop within another loop.
        2. A practical way to think about it is creating a for loop that counts years
            a. Then nesting a for loop that counts months within the above parent loop
            b. Result would be something like:
                i. Year
                    1. Month
                    2. Month
                    3. Month
                ii. Year
                    1. Month
                    2. Month
                    3. Month
        3.
           ```
           For (var year = 2023; year < 2025; year++) {
                   console.log(year);
                   For (var month = 6; month < 9; month++) {
                           console.log(“-----”, month)
                   }
           }
           ```
d. Examples of loops being used:
    i. If I'm coding an email client, I will get some structured data about the emails to be displayed in the inbox, then I'll use a loop to actually display it in a nicely-formatted way.

ii. If I'm coding an e-commerce site selling cars, I will get a source of nicely-structured data on each of the cars, then loop over that data to display it on the screen.
iii. If I'm coding a calendar online, I'll loop over the data contained in each of the days to display a nicely-formatted calendar.
iv. There are many, many other examples of using loops in code.
v. Using loops with data that is properly formatted for a given task is a crucial component of building software.
vi. In the lessons that follow, we'll learn about different ways of grouping related data and of displaying it on the screen using JavaScript.
vii. When combined with what you've already learned about loops, this gives you the skills to build various kinds of user interfaces where there is repetitive information.
viii. Some more specific examples include:
    1. looping over blog post titles in some structured data, and displaying each blog post title on a blog home page
    2. looping over social media posts in some structured data, and displaying each social media post based on some conditions
    3. looping over some structured data on clothing available for sale in an online clothing store, and displaying relevant data for each item of clothing

e. Additional resources:
    i. Comparison Operators
    ii. Truthy
    iii. Falsy
    iv. Conditional statements
    v. In JavaScript, there is also a shorthand version of writing a conditional statement, known as the conditional (ternary) operator: Conditional (ternary) operator

22. Functions
    a. One of the basic principles of programming can be summed up to DRY; Don't Repeat Yourself. Thanks to functions you can avoid repetition
    b. With functions, you can take several lines of code that performs a set of related actions and then group them together under a single label.
    c. Then when you need to run the code that you've saved, you can just invoke or call the function
        i. You can run the code as many times as you want
    d. These values are known as function parameters and are placed inside the function definition.
    e. The values passed to functions are arguments
    f. Arguments are parameters
    g. Example
        i. Function addTwoNums(a,b) {
                Var c = a+b;

```
                console.log(c);
            }
```
    ii. To call the function and pass parameters 2,4:
        1. addTwoNums(2,4);

23. Storing data in arrays
    a. A collection of pieces of data organized into an object
    b. Array literal syntax = []
        i. Var train1 = [“car1”, “car2”, “car3”];
            1. Indices = [0, 1, 2]
                a. 0 = car1
                b. 1 = car2
                c. 2 = ca3
            2. console.log(train1[0]);
                a. Result = car1
24. Objects
    a. A way to assemble a collective set of variables under one label
    b. Adding a property is known as extending the object
    c. Key value pairs of variables
        i. Variable name becomes property key
        ii. Variable value becomes property value
    d. Three common ways to build objects:
        i. One of the most common ways of building an object in JavaScript is using the object literal syntax: {}.
            1. Var user = {}; // create an object
        ii. Sometimes, an entire object can be immediately built, using the object literal syntax, by specifying the object's properties, delimited as key-value pairs
            1. //creating an object with properties and their values
```
                var assistantManager = {
                rangeTilesPerTurn: 3,
                socialSkills: 30,
                streetSmarts: 30,
                health: 40,
                specialAbility: "young and ambitious",
                greeting: "Let's make some money"
            }
```
        iii.
    e. Meant to mimic real-world objects
        i. Example 1 (dot-notation):
            1. dog.legs = 4;
            2. Dog.sound = “Woof”;
        ii. Example 2 (object literal): dog = {
```
            legs: 4,
```

Sound: “Woof”,

}

f. Dot notation
    i. Placing a dot operator between the object name and the property
        1. Example - human.hands = 2;
        2. Can access properties of the object like so:
            a. console.log(object.property); // displays property’s value in dev console

g. Object Literals
    i. To access the object, we can console log the entire object:
        1. console.log(object); // displays the object in dev console

h. Bracket Notation
    i. Example:
        1. Var house = {};
           House[“rooms”] = 4;
           House[‘color’] = “pink;
           house[”priceUSD”] = 12345;
        2. console.log(house);
            a. // {rooms: 4, color: 'pink', priceUSD: 12345}

i. Arrays are objects
    i. That means that arrays also have some built-in properties and methods.
    ii. One of the most commonly used built-in methods on arrays are the push() and the pop() methods.

j. Math object
    i. JavaScript has handy built-in objects. One of these popular built-in objects is the Math object.
    ii. Here are some of the built-in number constants that exist on the Math object:
        1. The PI number: Math.PI which is approximately 3.14159
        2. The Euler's constant: Math.E which is approximately 2.718
        3. The natural logarithm of 2: Math.LN2 which is approximately 0.693
    iii. Rounding methods
        1. Math.ceil() - rounds up to the closest integer
        2. Math.floor() - rounds down to the closest integer
        3. Math.round() - rounds up to the closest integer if the decimal is .5 or above; otherwise, rounds down to the closest integer
        4. Math.trunc() - trims the decimal, leaving only the integer
    iv. Arithmetic and calculus methods
        1. Here is a non-conclusive list of some common arithmetic and calculus methods that exist on the Math object:
            a. Math.pow(2,3) - calculates the number 2 to the power of 3, the result is 8
            b. Math.sqrt(16) - calculates the square root of 16, the result is 4

c. Math.cbrt(8) - finds the cube root of 8, the result is 2
d. Math.abs(-10) - returns the absolute value, the result is 10
e. Logarithmic methods: Math.log(), Math.log2(), Math.log10()
f. Return the minimum and maximum values of all the inputs: Math.min(9,8,7) returns 7, Math.max(9,8,7) returns 9.
g. Trigonometric methods: Math.sin(), Math.cos(), Math.tan(), etc.

v. Random Method
   1. A part of the Math object that can generate a number between 0 and 0.99

vi. Ceil Method
   1. A part of the Math object that rounds a decimal up to the nearest integer

vii. Combining above methods
   1. build some code that will combine the two to return a random integer between 0 and 10. The first step is to create a variable and assign it the value of the math dot random method multiplied by 10. I need to pass this variable through the ceil method to ensure that an integer instead of a decimal value is created. So, I declare the variable rounded and assign it the value of the math dot ceil method containing the decimal variable inside the parentheses. Finally, I use the console dot log method to output the variable rounded. If I run the code, notice that a value between 0 and 10 is output to the console.

k. Iterable
   i. An iterable is any datatype that can be iterated over using a “for of” loop.
   ii. Arrays and Strings are Iterables
   iii. In the world of JavaScript, it can often be said that strings behave like arrays. Strings are array-like.
   iv. You can run a for loop over an array of letters or a string and get the same result

l. String cheat sheet
   i. String Functions
      1. Length of a string - “string.length”
      2. Read individual character in a string - “string.charAt(0);”
      3. The concat() method joins two strings - "Wo".concat("rl").concat("d"); // 'World'
      4. The indexOf returns the location of the first position that matches a character:
      5. The lastIndexOf finds the last match, otherwise it works the same as indexOf.
      6. The split method chops up the string into an array of sub-strings - "ho-ho-ho".split("-"); // ['ho', 'ho', 'ho']
      7. Upper and lower case:

                a. greet.toUpperCase(); // "HELLO, "
                b. greet.toLowerCase(); // "hello, "
    m. Object Methods
        i. The same way you can add key/value pairs of data to an object, you can make one of those values a function.
        ii. Remember, value of an key in an object is considered a property
        iii. When a function is a property of an object, it is then referred to as a method
            1. This is a function that can be accessed only through the JavaScript object that it is a member of.
    n. Additional resources
        i. JavaScript Functions
        ii. JavaScript Object Basics
        iii. typeof operator in JavaScript
        iv. Arrays are "list-like objects"
    o.

25. typeOf
    a. JavaScript operator that evaluates a parameter and returns the data type as a string
26. Bugs and Errors
    a. Bug
        i. A bug causes a program to run in an unintended way
    b. Error
        i. An error causes a program to stop running
        ii. Some of the most common error types:
            1. Syntax Error
            2. Type Error
            3. Reference Error
27. Try/Catch Blocks
    a. Try
        i. If a piece of code throws an error, it can get wrapped inside a try block.
    b. Catch
        i. Then you can catch the error with the catch block, and use it to do something
        ii. The catch block accepts something called an error which is an object
    c. Throw
        i. Using the throw keyword, you can force an error to be thrown from the try block to the catch block
        ii. It is important to remember that you can use the throw keyword outside the the try block, but it will not be possible to catch it.
    d. Structure
        i. The try block starts with the try keyword
            1. Inside of the curly braces, you place the code that you think will throw an error

        ii. Next is the catch block which catches the error that the try block produces. It begins with the catch keyword and in parenthesis, you have a built-in error object that you can name whatever you like
            1. Inside the curly braces, you place the code you would like to execute

28. Syntax, logical, and runtime errors
    a. Most common errors in JavaScript:
        i. ReferenceError
            1. A ReferenceError gets thrown when, for example, one tries to use variables that haven't been declared anywhere
        ii. SyyntaxError
            1. Any kind of invalid JavaScript code will cause a SyntaxError
        iii. TypeError
            1. A TypeError is thrown when, for example, trying to run a method on a non-supported data type
        iv. RangeError
            1. A RangeError is thrown when we're giving a value to a function, but that value is out of the allowed range of acceptable input values
    b. Other errors:
        i. AggregateError
        ii. Error
        iii. InternalError
        iv. URIError
29. Types of empty values
    a. Undefined
        i. In JS, there may be times you're building something that hasn't been clearly defined yet and so you can't assign a value to it
        ii. fortunately, there is a way to store it so that you can assign it later using the undefined data type.
        iii. The undefined data type can only hold one value, undefined
    b. Null
        i. Intentional absence of any object value
        ii. It is also the return value of some built-in JavaScript methods
    c. Empty strings
        i. Var name1 = '';
        ii. Var name2 = "";
30. Additional resources:
    a. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function
    b. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch
    c. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols

    d. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math
    e. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String
    f. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Errors
    g. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/null
    h. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined

31. Introduction to functional programming
    a. Functions and data are separate from one another
    b. Formal - functional programming cannot be used for everything
    c. Informal - functional programming can't be used for everything
32. Introduction to object-oriented programming
    a. Functions and data are grouped together under an object
    b. Can access object properties using 'this' and dot notation
    c. In JS, one of the most elegant ways to efficiently build new objects is by using classes
    d. OOP helps developers to mimic the relationship between objects in the real world. In a way, it helps you to reason about relationships between things in your software, just like you would in the real world.
        i. OOP also:
            1. Allows you to write modular code
            2. Makes your code more flexible
            3. Makes your code reusable
    e. The four fundamental OOP principles are:
        i. Inheritance
        ii. Encapsulation
        iii. Abstraction
        iv. Polymorphism
    f. The thing to remember about Objects is that they exist in a hierarchal structure. Meaning that the original base or super class for everything is the Object class, all objects derive from this class. This allows us to utilize the Object.create() method. to create or instansiate objects of our classes.
    g. Inheritance
        i. There is a base class of a "thing".
        ii. There is one or more sub-classes of "things" that inherit the properties of the base class (sometimes also referred to as the "super-class")
        iii. There might be some other sub-sub-classes of "things" that inherit from those classes in point 2.

    iv. Note that each sub-class inherits from its super-class. In turn, a sub-class might also be a super-class, if there are classes inheriting from that sub-class.
    v. All of this might sound a bit "computer-sciency", so here's a more practical example:
        1. There is a base class of "Animal".
        2. There is another class, a sub-class inheriting from "Animal", and the name of this class is "Bird".
        3. Next, there is another class, inheriting from "Bird", and this class is "Eagle".
    vi. Thus, in the above example, I'm modelling objects from the real world by constructing relationships between Animal, Bird, and Eagle. Each of them are separate classes, meaning, each of them are separate blueprints for specific object instances that can be constructed as needed.
    vii. To setup the inheritance relation between classes in JavaScript, I can use the extends keyword, as in class B extends A.

h. Encapsulation
    i. In the simplest terms, encapsulation has to do with making a code implementation "hidden" from other users, in the sense that they don't have to know how my code works in order to "consume" the code.
    ii. I don't really need to worry or even waste time thinking about how the toUpperCase() method works. All I want is to use it, since I know it's available to me. Even if the underlying syntax - that is, the implementation of the toUpperCase() method changes - as long as it doesn't break my code, I don't have to worry about what it does in the background, or even how it does it.

i. Abstraction
    i. Abstraction is all about writing code in a way that will make it more generalized.
    ii. The concepts of encapsulation and abstraction are often misunderstood because their differences can feel blurry.
        1. An abstraction is about extracting the concept of what you're trying to do, rather than dealing with a specific manifestation of that concept.
        2. Encapsulation is about you not having access to, or not being concerned with, how some implementation works internally.

            iii. While both the encapsulation and abstraction are important concepts in OOP, it requires more experience with programming in general to really delve into this topic.

    j. Polymorphism
        i. Polymorphism is a word derived from the Greek language meaning "multiple forms". An alternative translation might be: "something that can take on many shapes".
        ii. So, to understand what polymorphism is about, let's consider some real-life objects:
            1. A door has a bell. It could be said that the bell is a property of the door object. This bell can be rung. When would someone ring a bell on the door? Obviously, to get someone to show up at the door.
            2. Now consider a bell on a bicycle. A bicycle has a bell. It could be said that the bell is a property of the bicycle object. This bell could also be rung. However, the reason, the intention, and the result of somebody ringing the bell on a bicycle is not the same as ringing the bell on a door.
            3.
        iii. To reiterate, polymorphism is useful because it allows developers to build objects that can have the exact same functionality, namely, functions with the exact same name, which behave exactly the same. However, at the same time, you can override some parts of the shared functionality or even the complete functionality, in some other parts of the OOP structure.

33. Constructors
    a. Javascript has a number of built in object types
        i. `Math`, `Date`, `Object`, `Function`, `Boolean`, `Symbol`, `Array`, `Map`, `Set`, `Promise`, `JSON`, etc.
            1. These objects are sometimes referred to as “native objects”
    b. Constructor functions, commonly referred to as just “constructors”, are special functions that allow us to build instances of these built-in native objects. All the constructors are capitalized
    c. Not all the built-in objects come with a constructor function
    d. Can build new strings as objects:
        i. Let apple = new String(“apple”);
        ii. The apple variable is an object of string type
    e. The most performant way is to use string literals, a primitive javascript value:
        i. Let pear = “pear”

    ii. The pear value, being a primitive value, will always be more performant than the apple variable which is an object

f. Besides being more performant, due to the fact that each object in JavaScript is unique, you can't compare a String object with another String object, even when their values are identical.
g. In other words, if you compare `new String('plum') === new String('plum')`, you'll get back `false`, while `"plum" === "plum"` returns true. You're getting the `false` when comparing objects because it is not the values that you pass to the constructor that are being compared, but rather the memory location where objects are saved.
h. Besides not using constructors to build object versions of primitives, you are better off not using constructors when constructing plain, regular objects.
i. Instead of `new Object`, you should stick to the object literal syntax: `{}`.
j. A RegExp object is another built-in object in JavaScript. It's used to **pattern-match strings** using what's known as "Regular Expressions". Regular Expressions exist in many languages, not just JavaScript.
k. In JavaScript, you can built an instance of the RegExp constructor using `new RegExp`.
l. Alternatively, you can use a pattern literal instead of RegExp. Here's an example of using `/d/` as a pattern literal, passed-in as an argument to the `match` method on a string.
    i. `"abcd".match(/d/); // null`
    ii. `"abcd".match(/a/); // ['a', index: 0, input: 'abcd', groups: undefined]`
m. Instead of using `Array`, `Function`, and `RegExp` constructors, you should use their array literal, function literal, and pattern literal varieties: `[]`, `() {}`, and `/()/`.
n. However, when building objects of other built-in types, we can use the constructor. Here are a few examples:
    i. `new Date();`
    ii. `new Error();`
    iii. `new Map();`
    iv. `new Promise();`
    v. `new Set();`
    vi. `new WeakSet();`
    vii. `new WeakMap();`

34. Inheritance
    a. Revolves around the prototype
    b. Prototype
        i. Often referred to as an original model from which other forms are developed
        ii. In JavaScript, the prototype is an object that can hold the properties to be shared by multiple other objects. And this is the basis of how inheritance works in JavaScript
        iii. This is why it's sometimes said that JavaScript implements a prototype of inheritance model.
    c. Javascript starts from the object itself when looking for properties to work with, Then if it can't find it on the object, it looks up to its prototype.
    d. It is important to remember that it doesn't look further if it finds the property on the immediate object. This makes for a simple mechanism for overriding inherited properties.
35. Creating Classes
    a. By now, you should know that inheritance in JavaScript is based around the prototype object.
    b. All objects that are built from the prototype share the same functionality.
    c. When you need to code more complex OOP relationships, you can use the **`class`** keyword and its easy-to-understand and easy-to-reason-about syntax.
    d. Imagine that you need to code a **`Train`** class.
        i. Once you've coded this class, you'll be able to use the keyword **`new`** to instantiate objects of the **`Train`** class.
        ii. For now though, you first need to define the **`Train`** class, using the following syntax:
            1.
```
class Train {}
```
        iii. So, you use the **`class`** keyword, then specify the name of your class, with the first letter capitalized, and then you add an opening and a closing curly brace.
        iv. In between the curly braces, the first piece of code that you need to define is the **constructor**:
            1.
```
class Train {
constructor() {
        }
}
```
        v. The **`constructor`** will be used to build properties on the future object instance of the **`Train`** class.

vi. For now, let's say that there are only two properties that each object instance of the `Train` class should have at the time it gets instantiated: `color` and `lightsOn`.

   1.
```
class Train {
constructor(color, lightsOn) {
this.color = color;
this.lightsOn = lightsOn;
        }
}
```

vii. Notice the syntax of the constructor. The constructor is a special function in my `Train` class.

viii. First of all, notice that there is no `function` keyword. Also, notice that the keyword `constructor` is used to define this function. You give your `constructor` function parameters inside an opening and closing parenthesis, just like in regular functions. The names of parameters are `color` and `lightsOn`.

ix. Next, inside the `constructor` function's body, you assigned the passed-in `color` parameter's value to `this.color`, and the passed-in `lightsOn` parameter's value to `this.lightsOn`.

x. What does this `this` keyword here represent?

   1. **It's the future object instance of the `Train` class.**
   2. Essentially, this is all the code that you need to write to achieve two things:
      a. This code allows me to **build new instances of the `Train` class.**
      b. Each object instance of the `Train` class that I build will have its own custom properties of `color` and `lightsOn`.

xi. Now, to actually build a new instance of the `Train` class, I need to use the following syntax:

   1.
```
new Train()
```

xii. Inside the parentheses, you need to pass values such as `"red"` and `false`, for example, meaning that the `color` property is set to `"red"` and the `lightsOn` property is set to `false`.

   1. And, to be able to interact with the new object built this way, you need to assign it to a variable.
   2. Putting it all together, here's your first train:
      a.
```
var myFirstTrain = new Train('red', false);
```

xiii. You can continue building instances of the **`Train`** class. Even if you give them exactly the same properties, they are still separate objects.

1. `var mySecondTrain = new Train('blue', false);`
2. `var myThirdTrain = new Train('blue', false);`

xiv. You can also add methods to classes, and these methods will then be shared by all future instance objects of my **`Train`** class.

1. Example

```
a. class Train {
b.     constructor(color, lightsOn) {
c.         this.color = color;
d.         this.lightsOn = lightsOn;
e.     }
f.     toggleLights() {
g.         this.lightsOn = !this.lightsOn;
h.     }
i.     lightsStatus() {
j.         console.log('Lights on?', this.lightsOn);
k.     }
l.     getSelf() {
m.         console.log(this);
n.     }
o.     getPrototype() {
p.         var proto = Object.getPrototypeOf(this);
q.         console.log(proto);
r.     }
s. }
```

2. Now, there are four methods on your **`Train`** class: **`toggleLights()`**, **`lightsStatus()`**, **`getSelf()`** and **`getPrototype()`**.
   a. The **`toggleLights`** method uses the logical not operator, **`!`**. This operator will change the value stored in the **`lightsOn`** property of the future instance object of the **`Train`** class; hence the **`!this.lightsOn`**. And the **`=`** operator to its left means that it will get assigned to **`this.lightsOn`**, meaning that it will become the new value of the **`lightsOn`** property on that given instance object.
   b. The **`lightsStatus()`** method on the **`Train`** class just reports the current status of the **`lightsOn`** variable of a given object instance.

   c. The **`getSelf()`** method prints out the properties on the object instance it is called on.
   d. The **`getPrototype()`** console logs the prototype of the object instance of the **`Train`** class. The prototype holds all the properties shared by all the object instances of the **`Train`** class. To get the prototype, you'll be using JavaScript's built-in **`Object.getPrototypeOf()`** method, and passing it **`this`** object - meaning, the object instance inside of which this method is invoked.

3. Now you can build a brand new train using this updated **`Train`** class:
   a. `var train4 = new Train('red', false);`
4. And now, you can run each of its methods, one after the other, to confirm their behavior:
   a. `train4.toggleLights(); // undefined`
   b. `train4.lightsStatus(); // Lights on? true`
   c. `train4.getSelf(); // Train {color: 'red', lightsOn: true}`
   d. `train4.getPrototype(); // {constructor: f, toggleLights: f, ligthsStatus: f, getSelf: f, getPrototype: f}`
      i. The result of calling **`toggleLights()`** is the change of true to false and vice-versa, for the **`lightsOn`** property.
      ii. The result of calling **`lightsStatus()`** is the console logging of the value of the **`lightsOn`** property.
      iii. The result of calling **`getSelf()`** is the console logging the entire object instance in which the **`getSelf()`** method is called. In this case, the returned object is the **`train4`** object. Notice that this object gets returned only with the properties ("data") that was build using the **`constructor()`** function of the **`Train`** class. That's because all the methods on the **`Train`** class do not "live" on any of the instance objects of the **`Train`** class - instead, they live on the prototype, as will be confirmed in the next paragraph.
      iv. Finally, the result of calling the **`getPrototype()`** method is the console logging of all the properties on the **`prototype`**. When the **`class`** syntax is used

in JavaScript, this results in **only shared methods being stored on the prototype**, while the **`constructor()`** function sets up the mechanism for saving instance-specific values ("data") at the time of object instantiation.

5. Thus, in conclusion, the class syntax in JavaScript allows us to clearly separate individual object's data - which exists on the object instance itself - from the shared object's functionality (methods), which exist on the prototype and are shared by all object instances. However, this is not the whole story.
6. It is possible to implement polymorphism using classes in JavaScript, by inheriting from the base class and then overriding the inherited behavior. To understand how this works, it is best to use an example.
    a. In the code that follows, you will observe another class being coded, which is named **`HighSpeedTrain`** and inherits from the **`Train`** class.
    b. This makes the **`Train`** class a base class, or the super-class of the **`HighSpeedTrain`** class. Put differently, the **`HighSpeedTrain`** class becomes the sub-class of the **`Train`** class, because it inherits from it.
    c. To inherit from one class to a new sub-class, JavaScript provides the **`extends`** keyword, which works as follows:

    ```
    class HighSpeedTrain extends Train {
    }
    ```

    d. As in the example above, the sub-class syntax is consistent with how the base class is defined in JavaScript. The only addition here is the **`extends`** keyword, and the name of the class from which the sub-class inherits.
    e. Now you can describe how the **`HighSpeedTrain`** works. Again, you can start by defining its constructor function:

    ```
    class HighSpeedTrain extends Train {
    constructor(passengers, highSpeedOn, color,
    lightsOn) {
    super(color, lightsOn);
    this.passengers = passengers;
    this.highSpeedOn = highSpeedOn;
    }
    }
    ```

f. Notice the slight difference in syntax in the constructor of the **`HighSpeedTrain`** class, namely the use of the **`super`** keyword.
g. In JavaScript classes, **`super`** is used to specify what property gets inherited from the super-class in the sub-class.
h. In this case, I choose to inherit both the properties from the **`Train`** super-class in the **`HighSpeedTrain`** sub-class.
i. These properties are **`color`** and **`lightsOn`**.
j. Next, you add the additional properties of the HighSpeedTrain class inside its constructor, namely, the passengers and highSpeedOn properties.
k. Next, inside the constructor body, you use the **`super`** keyword and pass in the inherited **`color`** and **`lightsOn`** properties that come from the **`Train`** class. On subsequent lines you assign **`passengers`** to **`this.passengers`**, and **`highSpeedOn`** to **`this.highSpeedOn`**.
l. Notice that in addition to the inherited properties, you also **automatically inherit** all the methods that exist on the **`Train`** prototype, namely, the **`toggleLights()`**, **`lightsStatus()`**, **`getSelf()`**, and **`getPrototype()`** methods.
m. Now let's add another method that will be specific to the **`HighSpeedTrain`** class: the **`toggleHighSpeed()`** method.

```
class HighSpeedTrain extends Train {
constructor(passengers, highSpeedOn, color,
lightsOn) {
super(color, lightsOn);
this.passengers = passengers;
this.highSpeedOn = highSpeedOn;
}
toggleHighSpeed() {
this.highSpeedOn = !this.highSpeedOn;
console.log('High speed status:',
this.highSpeedOn);
}
}
```

n. Additionally, imagine you realized that you don't like how the **`toggleLights()`** method from the super-class works, and you want to implement it a bit differently in the sub-class. You can add it inside the **`HighSpeedTrain`** class.

```
class HighSpeedTrain extends Train {
constructor(passengers, highSpeedOn, color, lightsOn) {
super(color, lightsOn);
this.passengers = passengers;
this.highSpeedOn = highSpeedOn;
}
toggleHighSpeed() {
this.highSpeedOn = !this.highSpeedOn;
console.log('High speed status:', this.highSpeedOn);
}
toggleLights() {
super.toggleLigths();
super.lightsStatus();
console.log('Lights are 100% operational.');
}
}
```

o. So, how did you override the behavior of the original **`toggleLights()`** method? Well in the super-class, the **`toggleLights()`** method was defined as follows:

```
toggleLights() {
this.lightsOn = !this.lightsOn;
}
```

p. You realized that the **`HighSpeedTrain`** method should reuse the existing behavior of the original **`toggleLights()`** method, and so you used the **`super.toggleLights()`** syntax to inherit the entire super-class' method.

q. Next, you also inherit the behavior of the super-class' **`lightsStatus()`** method - because you realize that you want to have the updated status of the **`lightsOn`** property logged to the console, whenever you invoke the **`toggleLights()`** method in the sub-class.

r. Finally, you also add the third line in the re-implemented **`toggleLights()`** method, namely:
   i. `console.log('Lights are 100% operational.');`

s. You've added this third line to show that I can combine the "borrowed" method code from the super-class with your own custom code in the sub-class.

t. Now you're ready to build some train objects.
   i. `var train5 = new Train('blue', false);`
   ii. `var highSpeed1 = new HighSpeedTrain(200, false, 'green', false);`

u. You've built the **`train5`** object of the **`Train`** class, and set its **`color`** to **`"blue"`** and its **`lightsOn`** to **`false`**.

v. Next, you've built the **`highSpeed1`** object to the **`HighSpeedTrain`** class, setting **`passengers`** to **`200`**, **`highSpeedOn`** to **`false`**, **`color`** to **`"green"`**, and lightsOn to false.

w. Now you can test the behavior of **`train5`**, by calling, for example, the **`toggleLights()`** method, then the **`lightsStatus()`** method:
   i. `train5.toggleLights(); // undefined`
   ii. `train5.lightsStatus(); // Lights on? True`

x. Here's the entire completed code for this lesson:
   i. `class Train {`
   ii. `constructor(color, lightsOn) {`
   iii. `this.color = color;`
   iv. `this.lightsOn = lightsOn;`
   v. `}`
   vi. `toggleLights() {`
   vii. `this.lightsOn = !this.lightsOn;`
   viii. `}`
   ix. `lightsStatus() {`
   x. `console.log('Lights on?', this.lightsOn);`
   xi. `}`
   xii. `getSelf() {`
   xiii. `console.log(this);`

```
}
getPrototype() {
var proto = Object.getPrototypeOf(this);
console.log(proto);
}
}

class HighSpeedTrain extends Train {
constructor(passengers, highSpeedOn, color, lightsOn) {
super(color, lightsOn);
this.passengers = passengers;
this.highSpeedOn = highSpeedOn;
}
toggleHighSpeed() {
this.highSpeedOn = !this.highSpeedOn;
console.log('High speed status:', this.highSpeedOn);
}
toggleLights() {
super.toggleLights();
super.lightsStatus();
console.log('Lights are 100% operational.');
}
}

var myFirstTrain = new Train('red', false);
console.log(myFirstTrain); // Train {color: 'red', lightsOn: false}
var mySecondTrain = new Train('blue', false);
var myThirdTrain = new Train('blue', false);

var train4 = new Train('red', false);
train4.toggleLights(); // undefined
```

xlv. `train4.lightsStatus(); // Lights on? true`

xlvi. `train4.getSelf(); // Train {color: 'red', lightsOn: true}`

xlvii. `train4.getPrototype(); // {constructor: f, toggleLights: f, ligthsStatus: f, getSelf: f, getPrototype: f}`

xlviii.

xlix. `var train5 = new Train('blue', false);`

l. `var highSpeed1 = new HighSpeedTrain(200, false, 'green', false);`

li.

lii. `train5.toggleLights(); // undefined`

liii. `train5.lightsStatus(); // Lights on? true`

liv. `highSpeed1.toggleLights(); // Lights on? true, Lights are 100% operational.`

y. Notice how the **`toggleLights()`** method behaves differently on the **`HighSpeedTrain`** class than it does on the **`Train`** class.

z. Additionally, it helps to visualize what is happening by getting the prototype of both the **`train5`** and the **`highSpeed1`** trains:

i. `train5.getPrototype(); // {constructor: ƒ, toggleLights: ƒ, lightsStatus: ƒ, getSelf: ƒ, getPrototype: ƒ}`

ii. `highSpeed1.getPrototype(); // Train {constructor: ƒ, toggleHighSpeed: ƒ, toggleLights: ƒ}`

aa. The returned values in this case might initially seem a bit tricky to comprehend, but actually, it is quite simple:

i. The prototype object of the **`train5`** object was created when you defined the class **`Train`**. You can access the prototype using **`Train.prototype`** syntax and get the prototype object back.

ii. The prototype object of the **`highSpeed1`** object is this object: **`{constructor: ƒ, toggleHighSpeed: ƒ, toggleLights: ƒ}`**. In turn this object has its own prototype, which can be found using the following syntax: **`HighSpeedTrain.prototype.__proto__`**.

    Running this code returns: **`{constructor: ƒ, toggleLights: ƒ, lightsStatus: ƒ, getSelf: ƒ, getPrototype: ƒ}`**.

7. Prototypes seem easy to grasp at a certain level, but it's easy to get lost in the complexity. This is one of the reasons why class syntax in JavaScript improves your developer experience, by making it easier to reason about the relationships between classes. However, as you improve your skills, you should always strive to understand your tools better, and this includes prototypes. After all, JavaScript is just a tool, and you've now "peeked behind the curtain".
8. In this reading, you've learned the very essence of how OOP with classes works in JavaScript. However, this is not all.
9. In the lesson on designing an object-oriented program, you'll learn some more useful concepts. These mostly have to do with coding your classes so that it's even easier to create object instances of those classes in JavaScript.
10. **Using class instance as another class' constructor's property**
11. Consider the following example:
    a. `class StationaryBike {`
    b. `constructor(position, gears) {`
    c. `this.position = position`
    d. `this.gears = gears`
    e. `}`
    f. `}`
    g.
    h. `class Treadmill {`
    i. `constructor(position, modes) {`
    j. `this.position = position`
    k. `this.modes = modes`
    l. `}`
    m. `}`
    n.
    o. `class Gym {`
    p. `constructor(openHrs, stationaryBikePos, treadmillPos) {`
    q. `this.openHrs = openHrs`

r. `this.stationaryBike = new StationaryBike(stationaryBikePos, 8)`

s. `this.treadmill = new Treadmill(treadmillPos, 5)`

t. `}`

u. `}`

v.

w. `var boxingGym = new Gym("7-22", "right corner", "left corner")`

x.

y. `console.log(boxingGym.openHrs) //`

z. `console.log(boxingGym.stationaryBike) //`

aa. `console.log(boxingGym.treadmill) //`

bb. In this example, there are three classes defined: **`StationaryBike`**, **`Treadmill`**, and **`Gym`**.

cc. The **`StationaryBike`** class is coded so that its future object instance will have the **`position`** and **`gears`** properties. The **`position`** property describes where the stationary bike will be placed inside the gym, and the **`gears`** propery gives the number of gears that that stationary bike should have.

dd. The **`Treadmill`** class also has a position, and another property, named **`modes`** (as in "exercise modes").

ee. The **`Gym`** class has three parameters in its constructor function: **`openHrs`**, **`stationaryBikePos`**, **`treadmillPos`**.

ff. This code allows me to instantiate a new instance object of the Gym class, and then when I inspect it, I get the following information:

   i. the **`openHrs`** property is equal to **`"7-22"`** (that is, 7am to 10pm)

   ii. the **`stationaryBike`** property is an object of the **`StationaryBike`** type, containing two properties: **`position`** and **`gears`**

   iii. the **`treadmill`** property is an object of the **`Treadmill`** type, containing two properties: **`position`** and **`modes`**

36. Designing an OOP
    a. Default Parameters
        i. ES6 feature
    b. There are two keywords that are essential for OOP with classes in JS:
        i. Extends - this keyword allows me to inherit from an existing class

      ii. Super - this keyword allows me to “borrow” functionality from a super-class, in a sub class
   c. Now I can start thinking about how to implement my OOP class hierarchy.
      i. Before I even begin, I need to think about things like: * What should go into the base class of `Animal`? In other words, what will all the sub-classes inherit from the base class? * What are the specific properties and methods that separate each class from others? * Generally, how will my classes relate to one another?

37. De-structuring arrays and objects
   a. Destructuring - copying an item, which then becomes independent of the original
   b. Built-in methods - these receive an object as its parameter
      i. Object.keys() - generates a string of object keys
      ii. Object.values() - generates a string of object values
      iii. Object.entries() - generates a string of object key/value pairs
   c. Loops
      i. For of loop
      ii. For in loop
38. Template literals
   a. Essentially, using template literals allows programmers to embed variables directly in between the backticks, without the need to use the + operator and the single or double quotes to delimit string literals from variables
39. Data Structures
   a. Suppose you receive some data on students’ test results and your task is to write a program that outputs an average grade from all the tests based on the raw data. Before you can code this task, you need to consider two separate issues
      i. How do you represent the given data in your app?
      ii. How do you code a solution to your task?
   b. JavaScript is somewhat limited to the types of data structures available, compared to other programming languages, like Python or Java.
   c. Most common data structures:
      i. Objects
         1. Unordered, non-iterable collection of key/value pairs.
         2. You use objects when you need to store and later access a value under a key.
         3. With objects, keys can only be strings or symbols
      ii. Arrays
         1. An ordered, iterable collection of values
         2. You use arrays when you need to store and later access a value under an index.
         3. We do not specify the index, JavaScript does this automatically
         4. You only use the index to access the specific value stored in the array.
         5. When working with arrays, it’s common to use a loop such as a for-loop to access and edit the data.

        iii. Maps
            1. Similar to an array because it’s iterable
            2. However, it consists of key-value pairs
            3. It is important not to confuse a map with an object
            4. With maps, any value can be used as a key
        iv. Sets
        v. This is another collection where each item in the collection must be unique.
        vi. For example, if you try to add a non-unique item to a set, this operation will simply not be run
    d. Array Methods
        i. forEach - loop over each of the array’s members
        ii. Filter - filters your arrays based on a specific test (condition?). Those array items that pass the test are returned.
        iii. Map - This method is used to map each array item over to another array's item, based on whatever work is performed inside the function that is passed-in to the map as a parameter.
    e. A map can feel very similar to an object in JS.
        i. However, it doesn't have inheritance. No prototypes! This makes it useful as a data storage.
    f. Other data structures in JavaScript
        i. Queues
        ii. Linked Lists (singly-linked and doubly-linked)
        iii. Trees
        iv. Graphs
    g. Operators
        i. Spread operator (...) - three dots before the object being spread
            1. Benefit - you don’t need to list each individual member of the array you want to pass to your function
            2. The spread operator allows you to pass all array elements into a function without having to type them all individually.
        ii. Rest operator
            1. Compared to spread operator which is used to ‘unpack a box’, the rest operator is used to build a smaller box and pack the items into it
            2. The rest operator allows you to take items from an array and use them to create a separate sub-array.
        iii. Both spread and rest operators have the same syntax in JavaScript, but they perform different functionalities. The spread operator in JavaScript expands values in arrays and strings into individual elements, whereas the rest operator puts the values of user-specified data into a JavaScript array.
    h. Destructuring
40. Javascript in the browser

a. Javascript modules - standalone units of code that you can reuse again and again
    i. You can:
        1. Add modules to programs
        2. Remove modules
        3. Replace modules
    ii. In all versions of JS, all functions defined on the window object are global
        1. While useful for simple projects, this created some issues when third party libraries or multiple developers became involved. For example, a global function from one script could override a function from another one using the same variable name. Techniques were developed to bypass these issues but they were not without flaws.
        2. For a long time, JS lacked built in natively supported module functionality
        3. An engineer at Mozilla tried to fix this through a project called Server JS which was later renamed to Common JS.
        4. Common JS is deisned to specify how modules should work outside of the browser environment
        5. It is mostly used on server side JS, namely, node.js.
        6. A downside of CommonJS is that browsers don't understand its syntax
b. <script type="text/javascript">
            console.log("Hello world");
    </script>
    i. You can also just use the script tag and it will automatically recognize the JS code
c. <script type="module">
d. CORS - Cross Origin Resource Sharing - a built in browser security feature
e. Javascript DOM manipulation
    i. Think of the DOM as a tv remote that lets you change the webpage content on the screen
        1. It even allows you to only replace certain parts of the page.
        2. The DOM give you a much finer-grained control than a TV remote ever could
        3. The DOM allows you to change properties of objects on a webpage
        4. The DOM is like a superpower remote for websites
    ii. The DOM is in the form of a JavaScript object with nested objects for different parts of the page.
        1. These objects have nested objects of their own until the entire HTML file is mapped out in what looks like a tree structure
        2. The DOM is the model of the HTML file saved as a JavaScript object in your browser's memory

3. The browser automatically builds the DOM for every webpage that it downloads

iii. Dev Tools

1. As a developer, you can interact with the page's DOM to make changes to the web page.
2. To interact with the DOM, you can use the Elements tab inside the browser's Dev Tools.
3. You get to the Dev Tools by right-clicking anywhere in a browser window and then clicking "Inspect".
4. Elements Tab
    a. The elements tab allows you to interact with the DOM of the currently active webpage using a graphical user interface, also known as a GUI.
5. Console Tab
    a. The browser also allows you to interact with the DOM using JavaScript. To do this, you should click the Console Tab in the browser's Dev Tools.
    b. You can focus on the Console panel by pressing the Escape key on your keyboard. Once done you can start typing JavaScript commands to view and manipulate the DOM. This is similar to how you interact with the DOM using the Elements panel, only this time you're using code to do it.
    c. The entire DOM object is saved inside the document variable and accessible via the console.
    d. It's important to remember that any changes I make to the DOM are relative to my browser's local copy of the webpage. I'm not updating the content of the live website so don't worry, you're not going to break it. If I reload the webpage, all changes I make to the DOM will be lost as it will reset to the page that was downloaded from the server.

iv. Element is singular for ID and plural for class name

f. Moving data around the web

i. JSON - Douglas Crockford came up with this data interchange format based on JS objects. The name give to this format is JSON, JavaScript Object Notation

ii. XML - before JSON the most common data interchange format was XML (Extensibile Markup Language). Due to XM"s syntax, it required more characters to describe the data that was sent. Also, since it was a specific stand-alone language, it wasn't as easily inter-operable with JS

iii. JSON format became the dominant data interchange format:

1. Its very lightweight, with syntax very similar to a "stringified JavScript object."
2. It's easier to handle JSON in JS since it is JS

41. JavaScript Object Notation - JSON
    a. Convert JSON strings to JS
    b. Use stringify to convert JS objects to JSON
    c. If you work with retrieving data from APIs converting the JSON strings to JS objects will be a standard workflow. You can then easily access the objects properties programmatically, This is a vital tool in your tool belt, encouraged to practice this.
42. JavaScript Environments
    a. Node
    b. NPM
43. NodeJS Environments
    a. Command Line
    b. Desktop Application
    c. Back-End
44. NPM package can also be called an NPM Module
45. When you want to start a new project, open a new folder and run 'npm init' to install the package
    a. Examples of libraries you can install
        i. React
        ii. Webpack
        iii. Bootstrap-vue
        iv. angular/core
46. JavaScript Testing Frameworks
    a. The test syntax itself becomes expectation documenting
    b. Tests as expectation documenting code - code syntax that specifies the expected result of passing specific values to your functions
    c. When tests fail, we say they are red. When tests pass, we say they are green. We also say tests are passing or failing
    d. Red-green-refactor cycle
        i. When testing your code, you may get some red and some green results. The red results will point to where you can make improvements.
        ii. The process of refining your code in response to red test results
        iii. This cycle is the basis of Test Driven Development (TDD)
47. Refactoring
    a. Changing code structure without changing its functionality
48. TDD approach
    a. Write failing test
    b. Re-write code to pass
    c. Optimize code without changing its results
49. Types of testing
        i. End to end testing - e2e
            1. Example: taking some laptops off the production line, opening them up, booting them, and making sure they work entirely as they should by acting like a normal user. More specific to web

development, end to end testing is trying to simulate what it would be like to be an end user that is using your app
2. Slowest form of testing; takes the most time to setup and action
3. E2E testing frameworks
    a. WebdriverJS
    b. Protractor
    c. Cypress

ii. Integration testing
1. Testing how parts of your system interact with other parts of your system. In other words, its testing how separate parts of your app work together
2. Integration testing software
    a. React testing library
    b. Enzyme
3. Faster and cheaper than E2E testing; but not as fast or cheap as Unit Testing

iii. Unit testing
1. The process of testing the smallest units of your source code in isolation.
2. A unit is the smallest piece of code that you can test separately from the rest of the app
    a. Usually a function or a method
3. Self contained; meant to test code in isolation; preferably separate from the rest of your app
4. This makes unit tests fast to run and easy to write

50. Introduction to Jest
    a. JavaScript does not have built in testing functionality
        i. Testing frameworks like Jest can be used for languages that don't have built in testing functionality
    b. Jest framework features
        i. Code coverage
            1. A measure of what percentage of my code is covered by tests
            2. Even 100% code coverage doesn't mean that you have tested for every conceivable expectation. It just means that there are some expectations tested for each line of my code
            3. The higher the code coverage, the lower the chance of having unidentified bugs
            4. As a rule the higher the percentage of code coverage, the lower amount of time required to write new tests. This however depends on whether there are incomplete software requirements pending or if you are going to receive more requirements in the future.
        ii. Mocking

                1. Allows you to separate the code that you are testing from it's related dependencies. In other words, you can use the mocking features to make sure that your unit testing is stand-alone
                2. Some libraries, such as sign-on, focus specifically on mocking
        iii. Snapshot
                1. Snapshot testing - used by developers to verify that there are no regressions in the DOM
    c. Testing libraries
        i. Jasmine
        ii. Mocha
        iii. Karma
        iv. qUnit
        v. Jest
    d. Jest is a testing framework
        i. Bult by Meta
        ii. JavaScript-based
        iii. Allows you to test
                1. Babel
                2. Typescript
                3. Node
                4. Angular
                5. Vue
                6. And other various frameworks
        iv. Jest runs tests for code in your current project to verify the expected output
    e. TDD
        i. Streamlined process of writing code that will satisfy some requirements

51. Course 3: Version Control
    a. Interview with Laila Rizvi - Back-End SWE at Meta / IG
        i. Communication is the most important skill for collaborating with other developers to ensure that you're on the same page when you're building products. Also so that people are keeping track of each other's timelines, and the understanding of what the product requirements are consistent
        ii. Effective collaboration is important so that we can move cohesively together on large projects with people that have a wide range of skills. As engineers, we actually have to collaborate with one another a lot. When we build features together, we have to work in parallel with one another to design the best features for our users. We also have to collaborate with a lot of non-engineers a lot.
        iii. For example, when we built Instagram Live, we had to work with our product managers to figure out what we should build. We had to work with our user researchers to figure out what areas we should focus on to build the best products for our users. We had to work with our designers to figure out the right look and feel for our product. We had to work as

engineers to figure out what we can actually build in the timeline that we had.

    iv. Communication, is one of the most important skills for working with other developers. Learning how to give developers the right amount of context for what they're working on.
    v. Learning how to ruthlessly prioritize your work is also very important as a software engineer, there's always going to be an endless amount of things that you can do to improve your product. Learning which things are most important to unblock other developers or unblock yourself is the most important skill.
    vi. The last thing that's really important to learn as a software developer is to accurately estimate your products, when you first start out, it'll be a little bit tricky, but over time learning how to say how long the project is going to take and being able to explain the trade-offs is going to be very critical.
    vii. The skills you need as a software developer changes from company to company a little bit. When you're at a big company like Meta, engineers are much more specialized. Whereas if you're in a startup, you wear many different hats.
    viii. As an engineer at Meta, we have to learn to be able to give just the right amount of context to people for what we need to get done. Because we're working with people a little context on the work we're doing. There's so many engineers there. But if you're in a startup, if people generally have a little more context on what you're working on, but they might not be as specialized in that area, you have to do a little bit more learning potentially, and you have to maybe give less context to them.

b. What is Version Control?
    i. Version Control - A system that records all changes and modifications to files for tracking purposes. Developers also use the term Source Control or Source Code Management
        1. Keep track of changes
        2. Provide access to history
        3. Revert and roll back
    ii. Different types of changes (files)
        1. Add (adding files)
        2. Modify/update (modifying or updating a file)
        3. Delete (deleting a file)
    iii. Version Control Benefits
        1. Revision history
        2. Identity
        3. Collaboration
        4. Automation
        5. Efficiency
    iv. DevOps
        1. Set of practices philosophies, and tools that increase an organization’s ability to deliver applications or services to a high quality and velocity.

        2. Version control is a key tool in this process, and it is used not only to track all changes but also to aid in software quality release and deployments.

v. Many different Version Control Systems (VCS) Available
    1. Subversion
    2. Perforce
    3. AWS Code Commit
    4. Mercurial
    5. Git

vi. Two types / categories
    1. Centralized Version Control System (CVCS)
        a. Contains a server and a client
        b. advantages
            i. Easier to learn
            ii. More access controls
        c. Disadvantages
            i. Slower
    2. Distributed Version Control System (DVCS)
        a. Each machine is the server in a way
        b. Advantages
            i. With DVCS, you don't need to be connected to the server to add your changes or view a file's history. It works as if you are actually connected to the server directly but on your own local machine. You only ever need to connect to the server to pull down the latest changes or to push your own changes. It essentially allows users to work in an offline state.
            ii. Speed
            iii. Performance

vii. A history of version control
    1. A long history going back to the 1980s
    2. One of the first significant Version Control Systems was the **Concurrent Versions System (CVS)**. It was first developed in 1986 by Walter F. Tichy at Purdue University and released publicly in 1990.
    3. **CVS** stores information about every file in a folder structure, including the name of the file, its location in the folder structure, who last modified it, and when it was last modified. The **CVS** also stores information about folders, including their names and who created them.
    4. It was popular for many years; however, it has some significant flaws in its design. **CVS** does not include integrity checks which means your data can become corrupted. When you update or submit changes to the system, if an error occurs, the system accepts the partial or corrupted files. Additionally, the system was designed mainly for text files, not binary files such as images or videos.
    5. The main successor to **CVS** was **Subversion (SVN)**.

viii. Staging vs. Production

1. Development Environments
    a. Every development team prior to releasing their new features or changes needs to verify that the code they do release is not going to cause any issues or bugs. In order to achieve this, they normally set up multiple environments for different ways to test and verify. A common practice is for teams to have a developer environment, a UAT or QA environment, and a staging environment. The main purpose of this flow is to find any potential issues that may arise due to changes or new features being added to the codebase. The more ways to test the changes the less likely bugs will be introduced.
2. Staging
    a. The staging environment should mimic your production environment. The reason for this is because you want to test the code in an environment that matches what you have in production. This allows teams to spot or find any potential issues prior to them getting to production. The closer the staging environment is to your production, the more accurate your testing is going to be. Staging environments can also be used for testing and verifying new features and allow other teams including QA or stakeholders to see and use those features as a pre-trial. Staging should also cover all areas of the architecture of the application including the database and any other services that may be required. Areas that benefit from staging environments include:
    b. New Features
        i. Developers submitting new features along with feature flags for turning them on and off should always do a testing round in a staging environment. They allow teams to verify that the feature works, it can be turned on and off via configuration flags and also that it does not break or interfere with existing functionality.
    c. Testing
        i. As the staging environment mimics your production environment, it's also a great place to run tests. QA teams will normally use it to verify new features, configuration changes or software updates/patching. The types of testing covered will be Unit testing, Integration testing and performance testing. All except performance testing can also be carried out in production. Performance can also be completed in production but only at specific times - usually out of hours as it will have a drastic effect on the user experience.

                ii. Sometimes it is not always feasible to have an exact replication either due to costs or time. Certain areas can be cut back - for example, if your service is load balanced on 10 virtual machines in production, you could still have 4 virtual machines in staging. The underlying architecture is the same but the overall performance may be different.

    d. Migrations
        i. Staging is a perfect place to test and verify data migrations. Snapshots can be taken from production and used to test your migration scripts to confirm your changes will not break anything. If in the case it does cause an issue, you simply rollback and try again. Doing something like a migration in production is extremely risky and error-prone.
    e. Configuration Changes
        i. Configuration can also cause headaches for teams, especially in a large cloud-based architecture. Having a staging environment will allow you to spot any potential issues or bottlenecks.

3. Production
    a. Production is live. It's out there for people to see and/or interact with. Any issues or problems you may have had should have been caught and fixed in the staging environment. The staging area gives the team a safety net to catch these possible issues. Any code that is deployed to production should have been tested and verified before the deployment itself.
    b. Downtime
        i. Downtime for any service especially customer facing will most likely be revenue impacting. If customers can not access or use your website or app to its full capabilities, it will most likely have a cost involved. Take for example an e-commerce company that allows users to buy goods and services online. If they release a new feature to their shopping cart which actually breaks the payment process, this will have an impact on customers not being able to buy goods online.
    c. Vulnerabilities
        i. Cyber-security should also play a big role in what gets released in production. Any updates to software such as patching or moving to the latest version should be checked and verified. This is also the same rule for not upgrading software when critical updates are released.

        d. Reputation
            i. Downtime or issues in production is damaging for a company as it does not instill confidence in end users. If something is down or broken it can cause the company to lose potential customers.

c. The Command Line
    i. Interact - Exchange information or send and receive information
    ii. Graphical User Interfaces (GUI) - facilitate your interactions with the computer or software
        1. GUIs are popular because they require very little training to use
        2. GUIs offer an easy way to interact with devices, but they also somewhat limit the scope of the human-computer interaction
    iii. By learning just a few commands, you can perform various tasks:
        1. Creating directories
        2. Creating files
        3. Combining directories
        4. Copying and moving files
        5. Performing advanced searches
    iv. As you become more advanced in using the command line, you will be able to perform tasks such as:
        1. Track software
        2. Access remote servers
        3. Search
        4. Unzip
        5. Access manuals (access and display in CLI)
        6. Install, upgrade, and uninstall software
        7. Mount and unmount computer drives
        8. Check disk space
        9. Automate
        10. Control user access
        11. Stop, start, and restart programs
        12. Create aliases of only a few characters long to initiate very long commands
        13. Download
        14. Run
        15. Containerization
    v. Basic commands
        1. Change Directory - cd
            a. Example”: cd ~/Desktop

            b. Move back to parent directory - cd ..
        2. Make a new file of whatever type you specify - touch
            a. Example: touch example.txt
        3. Make new folders - mkdir
            a. Example: mkdir new-folder-name

d. What are Unix commands?
    i. Manual - man
        1. Example: man ls
            a. This will give the detailed manual of instructions for the list command 'ls'.
        2. Man Pages
            a. When first learning commands from bash, it can feel a bit daunting. Luckily, every command has its manual (or man pages for short). The man page lists all the flags and options that a particular command has to offer.
    ii. Flags: used to modify the behavior of a command
        1. Can use these in conjunction with Unix commands
        2. Think of flags as options that can either change or extend the functionality of the given command
    iii. Print working directory - pwd
    iv. Copy - cp
    v. Move - mv
    vi. Editing
        1. There are many options for editing files in bash. The most common is usually VI or Vim. VI stands for visual editor. It's used for making edits and changes to a file and saving them. It's similar to what you may have used in applications like Word. VIM is a better version of VI with some improvements - hence its name: visual editor improved. Learning the commands in Vim will feel different from GUI applications, but once you practice, it will feel like second nature. Vim uses modes to determine the commands you can work with:
            a. Normal mode: Default mode
            b. Insert mode: Allows the contents of the files to be edited.
            c. Command line mode: Normal commands begin with :
    vii. Pipes
    viii. Redirection
        1. Standard Input

        2. Storing Input
        3. Standard Output
        4. AKA - stdin stdout and stderr
        5. Can use redirection to output terminal console to text file
    ix. Grep - Global regular expression print
        1. Used for searching across files and folders as well as the contents of files

e. Git and Github
    i. Connecting to Github
        1. HTTPS
        2. SSH
    ii. Git workflow: Active Directory > Staging Area > Commits > Remote Repository
    iii. Git commands
        1. Git init
        2. Git remote
        3. Git status
        4. Git add
        5. Git commit
        6. Git push
        7. Git pull
        8. Git branch
        9. Git stash
        10. Git revert
        11. Git restore
        12. Git diff
        13. Git blame
        14. Cat .git/HEAD
        15. git log --pretty=oneline
        16.
        17.
    iv.