1. About JavaScript
   1. JavaScript can be modified after running. For example, a button click can load more script, or modify (override) an existing function. You can also modify object structures at will. Also, nothing is strongly typed which means you can manipulate almost everything, at runtime
   2. a **dynamic programming language** is a class of high-level programming languages, which at runtime execute many common programming behaviours that static programming languages perform during compilation
   3. type checking done at runtime instead of at compile time
      1. Explain diff between runtime and compile time
      2. Also means variable of same name can be used to hold different data types
   4. JavaScript is a dynamically typed language, which means that data types of variables are determined by the value they hold at runtime and can change throughout the program as we assign different values to them
2. Values
   1. Bit - something that has two states
      1. On or off
      2. Zero or one
      3. High charge or low charge
      4. Strong or weak signal
      5. Shiny or dull spots
   2. Values
      1. chunks of bits that represent pieces of information
      2. Stored in memory locations
   3. Creating a value
      1. invoke its name
3. Data types and operators
   1. Loosely typed
      1. JavaScript determines type based on value
      2. Don’t need to name type specifically
      3. Problem with loosely typed - makes it easy to accidentally assign a value of the wrong type to a variable
         1. Won’t be caught at compile time, just run time
         2. Makes it harder to debug code
   2. Data types
      1. Definition - a classification of data that determines the kinds of values that a variable can store in programming
      2. Two categories:
         1. Non-primitive – three types (really one type)
            1. Object, Array, RegExp
      3. Primitive data types
         1. Also known as built-in data types
         2. Properties
            1. Directly stored in memory
            2. Copy by value (more on this later)
            3. Immutable

Value cannot be changed once assigned

Can assign new value but can’t change old value (I have example somewhere)

* + - * 1. stored in the stack space of the system (more on this later)
      1. Seven types - string, number, BigInt, boolean, symbol, undefined, null
      2. typeof operator
         1. a special keyword (operator) that will tell you what is the data type of a certain variable’s value (string, number, boolean, etc)
         2. Can check two ways:

With parentheses

testVariable = 1;  
variableTypeTest2 = typeof(testVariable);

without parentheses

variableTypeTest1 = typeof testVariable;

Brackets aren’t required because technically typeof is an operator, not a method

Brackets make it easier to read

* + - * 1. typeof null returns object, even though it is a primitive
        2. Empty values
        3. Null and Undefined - Difference between
      1. string
         1. Stores sequences of characters enclosed in single or double quotes
         2. How they are written

Can use single quotes, double quotes or backticks (template literal)

Problem with using single quotes - you can’t then use a single quote as an apostrophe

* + - * 1. Breaking up code lines within strings

Use newline character \n

let haiku = “Oh my aching back\n might as well claim workers comp \n It’s thrown out again”

* + - * 1. Escaping

Used for special characters (give examples)

Done with Backslash character \

* + - * 1. Concatenation

Use plus sign + to put two or more strings together, along with variables or numbers to go with the strings

\\ Concatenation  
let numOfStudents = 23;  
console.log(“There are “ + numOfStudents + “ students in my class.”);  
 \\ There are 23 students in my class.

* + - * 1. Template literals

Putting whole string in backticks instead of quotes

Using backticks lets you include JavaScript variables within string instead of breaking up string doing concatenation

Variables are included by putting in curly braces and prefixing braces with dollar sign

\\ Template literal  
console.log(`There are ${numOfStudents} students in my class.`);   
\\ There are 23 students in my class.

* + - 1. Number
         1. Numeric values
         2. Only one type - no integer vs float
         3. Fixed number of bits - 64

Can represent 2^64 different numbers (18 quintillion - an 18 with 18 zeros after it)

One bit has to represent negative sign

Some bits have to store position of decimal point

Maximum whole number - 9 quadrillion (15 zeros)

For scientific notation, add an e for exponent followed by exponent of the number   
2.998e8 = 2.998 x 10^8 = 299,800,000

* + - * 1. Number limits
        2. Number operators (is it talking about addition, subtraction and such? If so, belongs elsewhere)
        3. Precision

Integer

Calculations with whole numbers (integers) that are smaller than 9 quadrillion are guaranteed to always be precise

Floating (or is it fixed?)

Calculations with fractional numbers are generally not precise

Only problematic in specific situations

Treat fractional digital numbers as approximations, not precise values

* + - * 1. NaN
        2. Infinity/-Infinity 1.7976931348623157e+308
        3. Underlines in numbers for readability
        4. Numeric strings
        5. Adding numbers and strings
        6. https://codeburst.io/javascript-why-does-3-true-4-and-7-other-tricky-equations-9dd13cb2a92a
      1. BigInt
         1. For values too large to be represented by number primitive
         2. Two ways to create:

Add n to the end of an integer literal

call the BigInt() function (without the word new) and give it an integer or string value

* + - * 1. Differences from numbers

Cannot be used with methods from Math object

Cannot be mixed with a number value in operations

Must be coerced to the same type

Coercing values back and forth can cause loss of the precision of a BigInt value

* + - 1. Boolean
         1. Two values: true and false
         2. Coercion

Can coerce a non-boolean into a boolean with two exclamation marks or by calling the Boolean() function (more on this later)

Do not use the word new

When coerced:

undefined and null turn into false

0, -0, NaN turn into false; all other numbers turn into true

0n turns into false; other BigInts turn into true

The empty string “” turns into false; other strings turn into true

Symbols and all objects become true

Boolean coercion does not convert objects to primitives

Values that get coerced to false are called falsy values; all other values are truthy values

* + - 1. symbol
         1. New data type
         2. Used to specify that two variables are not equal, even though their value and type are the same
         3. Works with the following operators: +, -, \*, /, %, +=, -=, ++, —, <, >, =, ==, ===, !==, !=, <=, >=
         4. How used

let str1 = “JavaScript is fun!”;  
let str2 = “JavaScript is fun!”;  
console.log(“These two strings are the same: “, str1 === str2); // true  
let sym1 = Symbol(“JavaScript is fun!”);  
let sym2 = Symbol(“JavaScript is fun!”);  
console.log(“These two Symbols are the same: “, sym1 === sym2); // false

* + - * 1. Useful as properties of objects
      1. undefined
         1. definition - a variable that has not been assigned a value

But can assign value of undefined to variable - BAD PRACTICE

Would cause one variable that is undefined and another variable that is manually set to undefined to be considered equal

* + - * 1. typeof undefined variable returns “undefined”
      1. null
         1. definition - having no value or empty value

But it is a type of assignment value because you can assign it to a variable

typeof null returns “Object”

null !== undefined but null == undefined

null is case-sensitive - use lowercase

* + 1. Non-primitive
       1. also known as object references
       2. Properties
          1. Mutable

Value can be changed after assignment

* + - * 1. Copy by reference

stored in the heap memory of the system (more on this later)

* + - 1. three types (really one type) - Object, Array, RegExp (function???)
      2. Object
         1. Name/value pairs (also called key/value pairs)
         2. How written
         3. Looking up non-existent property in an object returns undefined

var test = {};  
console.log(test.prop); // undefined

* + - * 1. primitive data types have a wrapper that is an object version of the data type

this allows those data types to have methods upon which to call to perform certain operations (more on this later)

* + - 1. Array
         1. How written
         2. Zero-based
      2. RegExp
    1. Primitive data types have wrappers that are objects and that have methods you can run on them (more on this later)
  1. Copy by value vs copy by reference
     1. Copy by value - Means new variable holds a copy of the old variable
        1. var a = 1;  
           var b = a;  
           console.log(a); // 3  
           console.log(b); //1
     2. Copy by reference –
  2. Operators
     1. Binary – uses two values
        1. instanceof – discuss when talking about classes and prototypes
     2. Unary – uses one value
        1. +a : converts to number
        2. -a : converts to number and negates
        3. ++a : adds 1 to a and returns new a
        4. a++ : add 1 to a and returns original a
        5. --a : subtracts 1 from a and returns new a
        6. a-- : subtracts 1 from a and returns original a
     3. Ternary – uses three values
     4. Assignment Operators
     5. Arithmetic Operators
        1. Arithmetic operations take two number values and produce a new number from them
           1. Symbols like + and \* are called operators
           2. Putting an operator between two values will apply it to those values and produce new value
        2. Order in which operators are applied is determined by precedence of the operators
           1. Can change order of operations with parentheses
     6. Comparison Operators
     7. Logical Operators
     8. Type conversion
     9. How expressions are evaluated (left to right)
     10. Order of precedence
     11. Value Comparison
     12. == vs ===
     13. Practice Examples

1. Programming Structure
   1. Definitions
      1. expression - something that produces a value / a fragment of code that represents a single value / any valid piece of code that resolves to a value / a sentence fragment
         1. Every value written literally (22, “psychoanalysis) is an expression
            1. Includes expressions between parentheses, and binary and unary operators applied to an expression
            2. Expressions can contain other expressions
         2. two types – those that have side effects and those that purely evaluate
         3. resolves to a value – when everything a computer needs to do is done, some sort of value is produced
            1. the expression 1 + 1 “resolves to” 2 – it has the value of 2
            2. the expression x = 7 assigns its value (7) to the variable called x
         4. expressions are made up of operands (the number 1 or the variable x) and operators (= or +)
         5. operands can be any of the JavaScript data types, including objects and arrays
         6. programmers will assign values to variables and use those variables as operands
      2. statement - something that performs an action / a complete thought instructing the computer to do something / a full sentence
         1. made up of operands (which are like nouns) and operators (which are like verbs)
         2. You can turn an expression into a statement simply by adding a semicolon to the end of it
         3. A statement that doesn’t affect the world is useless
         4. In some cases, semicolon can be omitted, but BAD IDEA
   2. Bindings
      1. Variables - bindings that can represent different values each time the code runs
         1. A variable is a box with a name on it, and the value is what you find inside that box
         2. Can be assigned new value while code is running
         3. Must first declare variable then assign value to it
            1. Can declare and assign in one step
         4. Declare using var, let or const
            1. Don’t need to use in calling variable again
         5. Assigning values - can hardcode or can come dynamically from external input (is this best definition?)
         6. camelCase
      2. var
         1. Old method of declaring variable
         2. Function scope
         3. Can have new value assigned to it after having previous value assigned to it
      3. let
         1. Block scope
         2. Can have new value assigned to it after having previous value assigned to it
      4. const
         1. Block scope
         2. Cannot have new value assigned to it after having previous value assigned to it
            1. Will cause Uncaught TypeError: Assignment to constant variable
      5. Binding name rules
         1. Cannot contain spaces but can use underscores
         2. Should use camel case and be descriptive (age, not x)
            1. Exception: temporary variables for utility function such as incrementing in for loop
      6. Reserved words
   3. Functions
      1. Function definitions
      2. Invoking/calling/applying
      3. Methods
         1. A method is a function stored as a property of an object
      4. ‘this’ keyword
      5. Accessing methods
         1. Invoking/calling/applying
      6. Event
         1. Called
         2. Self-invoked
      7. Purpose of ()
         1. Accessing function without ()
      8. Arguments/Parameters
      9. Return values
      10. indentation
   4. Control flow
   5. Conditional execution
      1. If / else if / else
         1. If (If statement by itself)
            1. checks single condition

if condition is true, code does something

if condition is false, code does nothing

* + - * 1. A single condition can consist of multiple individual conditions connected by AND, OR and NOT operators
      1. if/else
         1. If statement with else statement but no else-if statements
         2. checks single condition

if condition is true, code does something

if condition is false, code does something else

* + - * 1. code will always do something
      1. If/else if
         1. If statement with one or more else-if statements but no else statement
         2. checks first condition

if condition is true, code does something

code then skips over the else-if statements and exits the block

if condition is false, code checks condition of first else-if statement

if that condition is true, code does something, skips over any other else-if statements and exits the block

if that condition is false, code goes to the next else-if and does the same thing

* + - * 1. if condition for if statement is false and all conditions for all else-if statements are false, code does nothing
      1. If/else if/else
         1. If statement with one or more else-if statements plus else statement
         2. checks if condition

if true, code does something

if the if condition is false, moves to first else-if condition and checks that

* + - * 1. each else-if condition is checked until one is found to be true

when an else-if condition is true, code does something, skips any other else-if statements plus the else statement and exits the block

* + - * 1. if none of the else-if conditions are true, code moves to else statement

no condition on else statement - code does something

* + - * 1. code will always do something
        2. If, else-if and else statements can be nested in each other
      1. With else-if statements, have to be careful to have them in correct order to get the result you want
         1. var gold = 54;  
            if (gold > 20) console.log(“I am comfortable”);  
            else if (gold > 50) console.log(“I am rich!!!”);  
              
            - even though gold > 50, you will get “I am comfortable” instead of “I am rich!!!”  
            - need to put:  
              
            if (gold > 50)  
            else if (gold > 20)  
              
            - will check first condition - condition will be false - fall through to second condition
    1. Switch statements
       1. another way to check multiple conditions and have code do something based on which condition is true
       2. case
       3. default
       4. break
          1. Unlike else if, with a switch statement multiple conditions could be found as true and have code executed if you fail to include break statement at end of each code section for the particular conditions

without break statement, execution will fall through to the next case

will eventually hit default and execute that, even if you don’t want it to

* + - 1. matching multiple cases
      2. strict comparison
         1. value = 4;  
            if (value == "4") {  
             console.log("it is 4");  
            else {  
             console.log('it is nothing');  
            }
         2. switch (value) {  
             case "2":  
             console.log('it is 2');  
             break;  
             case "4":  
             console.log('it is 4');  
             break;  
             default:  
             console.log('it is nothing');  
            }
  1. Loops
     1. For loops
        1. for
        2. for-in
        3. for-of
     2. while/do-while
        1. while
           1. may not run at all
        2. do-while
           1. will run at least once
     3. break/continue

1. Functions
   1. Syntax
   2. Defining functions
      1. Parameters
      2. Arguments
   3. Bindings and scope
      1. Global
      2. Local
         1. Function
         2. Block
      3. Nested scope
   4. Functions as values
   5. Function declaration
      1. Hoisting
         1. a JavaScript mechanism where variables and function declarations are moved to the top of their containing scope during compilation, before the code is executed
            1. Means you can use a variable or function before it’s declared in your code
            2. For variables, only declaration is moved to top, not initialization
         2. variables declared with var  
            Ex. 1  
            var x = 10;  
            console.log(x); // 10   
            - Executes in this order:  
            var x; // x is available but undefined  
            x = 10; // x is now defined  
            console.log(x);   
              
            Ex. 2  
            console.log(x); // undefined  
            var x = 10;   
            - Executes like this:  
            var x; // x is available but undefined  
            console.log(x);  
            x = 10; // x is now defined   
              
            Ex. 3  
            console.log(x); // ReferenceError: Can’t find variable: x   
            - This proves hoisting - In example 2, if declaration of x wasn’t hoisted, it would also return a reference error   
              
            Ex 4  
            var a = 10;  
            document.write(a);  
            var b = 20;   
            - Compile phase:  
            var a;  
            var b;  
            a = 10;  
            document.write(a);  
            b = 20;
         3. variables declared with let/const
            1. hoisted but not initialized  
                 
               Ex 1  
               console.log(x); // ReferenceError: Cannot access uninitialized variable  
               let x = 10;   
                 
               Ex 2  
               let x;  
               const y = 30;  
               console.log(x); // undefined  
               console.log(y); // 30  
               x = 40;  
               console.log(x); // 40
         4. Function declarations are fully hoisted, including the function body   
            Ex 1  
            sayHello(); // Hello!  
            function sayHello() { // function declaration  
             console.log(‘Hello!’);  
            };
         5. Function expressions are not hoisted   
            Ex 1  
            sayHi(); // TypeError: sayHi is not a function (In sayHi(), ‘sayHi’ is undefined)  
            var sayHi = function () { // function expression  
             console.log(‘Hi!’);  
            }   
              
            Ex 2  
            sayHi(); // ReferenceError: Cannot access uninitialized variable   
            let sayHi = function() { //   
             console.log(‘Hi!’);  
            }   
            - sayHi marked as error in code, with this message: ’sayHi’ was used before it was declared, which is illegal for ‘let’ variables
            1. Same with const
         6. Best practices
            1. Declare variables and functions before using them
            2. Use let and const to minimize unexpected hoisting behavior
            3. Avoid relying too heavily on hoisting for code readability
   6. Function scope vs Block scope
      1. Function scope - variable is confined to the function
         1. Cannot be read outside the function
         2. But is accessible anywhere within the function
      2. Block scope - variable is confined within the closest set of curly braces
         1. Cannot be read outside the function
         2. Cannot be read anywhere in the function that is outside of its enclosing curly braces  
              
            // Function Scope  
            function functionScopeExample() {  
             var depositAmount = 200;  
             if (depositAmount < 10000) {  
             var isInsured = true;  
             }  
             console.log(depositAmount); // 200  
             console.log(isInsured); // true  
            }  
              
            // Block Scope  
            function blockScopeExample() {  
             let depositAmount = 200;  
             if (depositAmount < 10000) {  
             let isInsured = true;  
             }  
             console.log(depositAmount); // 200  
             console.log(isInsured); // ReferenceError: Can't find variable: “isInsured"  
            }
      3. Lexical scope means a function can access variables from its outer (enclosing) scope
   7. arrow functions
      1. Arrow functions have an implicit return feature. If the function body consists of a single expression, you can omit the return keyword.  
           
         () => ‘foo’ is a function that takes no parameters and returns the string ‘foo’
      2. To user implicit return for an object literal, you have to wrap the object literal in parentheses.  
           
         const noop = () => { foo: ‘bar’ };  
         console.log(noop()); // undefined  
           
         const createFoo = () => ( { foo: ‘bar’ } );  
         console.log(createFoo()); // { foo: ‘bar’ };
   8. call stack
      1. last in, first out  
           
         function greet(name) {  
          console.log(`Hello, ${name}!`);  
         }  
           
         function welcome() {  
          console.log("Welcome!");  
         }  
           
         function main() {  
          greet("John");  
          welcome();  
         }  
           
         main();

* Here, when main() is called, it puts greet("John") on top of the stack
* After executing greet, it removes it and then adds welcome() to the top
* This stacking and unstacking happen in the order functions are called  
  1. optional arguments
  2. higher order functions
     1. Treat functions as values
        1. Assign them to another variable
        2. Pass them as arguments to another function
        3. Return them from another function
        4. To be treated as a value the function must be returning a value
     2. Higher order function is one that either takes another function as an argument or returns a function as an output
        1. Some higher-order functions built into JavaScript
           1. Array.prototype.map
           2. Array.prototype.filter
           3. Array.prototype.reduce
     3. Passing function as an argument
        1. Useful when you need many different functions that all have certain parts of their function in common with all the other functions
        2. Allows us to abstract over actions
           1. Put all the common parts into one function
           2. Pass the logic that differs between each function in as an argument to the higher function
           3. Analogy - making a sandwich

Could have three functions for making three different sandwiches

PB & J

Ham & Cheese

Egg Salad

Each function has certain steps in common

Get bread out of the cupboard

Get plate

Take to table

Open bread and put two slices on plate

Put something on bread

Close slices and eat

The three functions only differ in what is put on bread

Solution

make three functions that only get out the fixins

Make one function that has all the steps in common

Pass in one of the three functions as the fixins for the sandwich function

In the sandwich function, put whatever was passed in on the bread

* 1. Closure
     1. Gives you access to an outer function’s scope from an inner function
        1. Applies with nested functions
        2. Inner functions have access to variables declared in the outer function scope, even after the outer function has returned   
           const createSecret = (secret) => {  
            return {  
            getSecret: () => secret,  
            setSecret: (newSecret) => {  
            secret = newSecret;  
            },  
            };  
           };  
           const mySecret = createSecret(“My secret”);  
           console.log(mySecret.getSecret()); // My secret  
           mySecret.setSecret(“My new secret”);  
           console.log(mySecret.getSecret()); // My new secret
     2. Closure variables are live references to the outer-scoped variable, not a copy
        1. If you change outer-scoped variable, the change is reflected in the closure variable, and vice versa
           1. Means other functions declared in the same outer function will have access to the changes
     3. Uses for closures:
        1. Data privacy   
           // Data privacy  
           const createCounter = () => {  
            let count = 0;  
            return {  
            increment: () => ++count,  
            decrement: () => —count,  
            getCount: () => count,  
            };  
            };  
           };
        2. Curried functions and partial applications   
           // A curried function takes multiple arguments one at a time  
           const add = (a) => (b) => a + b;  
           // A partial application is a function that has been applied to some,   
           // but not yet all of its arguments   
           const increment = add(1); // partial application   
           increment(2); // 3
     4. Pure functions
        1. Used in functional programming
        2. Used because they are predictable
           1. Easier to understand, debug, and test than impure functions
        3. Follow two rules:
           1. Deterministic - given the same input, will always return the same output
           2. No side-effects - a side effect is any application state change that is observable  outside the called function other than its return value
        4. Examples of non-deterministic functions:
           1. A random number generator
           2. A global variable that can change state
           3. A parameter that can change state
           4. The current system time
        5. Examples of side effects:
           1. Modifying any external variable or object property (e.g. a global variable,   
              or a variable in the parent function scope chain)
           2. Logging to the console
           3. Writing to the screen, file, or network
           4. Throwing an error - instead the function should return a result indicating the error
           5. Triggering any external process
           6. Making an HTTP request
           7. Setting an async timer
           8. Mutating arguments to a function
           9. Generating random numbers or unique random IDs (such as Math.random() or Date.now() )
     5. Closures occur when a function retains access to variables from its outer scope, even after that outer function has finished executing
     6. This concept is pivotal for creating private variables and maintaining state across function calls  
          
        function outerFunction() {  
         let outerVar = "I’m from outside!";  
          
         function innerFunction() {  
         console.log(outerVar);  
         }  
          
         return innerFunction;  
        }  
          
        const closureExample = outerFunction();  
        closureExample(); // Prints: I’m from outside!
* In this example, innerFunction closes over the outerVar, allowing it to access outerVar even after outerFunction has completed its execution.
  1. recursion
  2. controlling function size
  3. functions and side effects

1. Objects and Arrays
   1. Data sets/arrays
      1. Creating
         1. Using [] for new Array
      2. Multiple lines
      3. Keyword “new”
      4. Accessing elements of an array
      5. Changing array elements
         1. Last element (length – 1)
      6. Accessing full array
      7. Looping through array
   2. Properties
      1. Name/value pairs
   3. Methods
      1. Function definitions
      2. ‘this’ keyword
      3. Accessing methods
   4. Objects
      1. Difference between arrays and objects
         1. Named indexes vs numbered indexes
      2. Object literals
      3. Accessing object members
   5. Mutability
      1. Identity
      2. Updating array immutably
         1. Copy array and then update the copy and return that   
            const arr = [‘a’, ‘b’]  
            arr.concat{‘c’) // mutates original array  
              
            const arr2 = arr.concat(‘c’) // immutable b/c copies array and updates the copy  
            const arr3 = arr.slice()  
            arr3.push(‘c’) // also immutable
   6. Array methods
      1. length
      2. pop/push
      3. substring
      4. shift/unshift
      5. indexOf/lastIndexOf
      6. forEach
      7. toString
      8. join
      9. delete
      10. concat
      11. sort
      12. compare function
      13. reverse
      14. Immutable array methods
          1. filter
          2. map
          3. reduce
   7. String methods
      1. slice/substring/substr
         1. differences
      2. indexOf/search
         1. differences
      3. trim
      4. split/join
      5. length
      6. toUpperCase/toLowerCase
      7. repeat
      8. replace
      9. concat
      10. padStart/padEnd
      11. charAt/charCodeAt
   8. Number methods
      1. toString
      2. toExponential
      3. toFixed
      4. toPrecision
      5. valueOf
      6. Number method
         1. parseInt
         2. parseFloat
   9. Rest parameters
   10. Math Object
       1. Namespace – definition
       2. Math functions
          1. min/max
          2. sqrt
          3. random
             1. Not really random
             2. Creates decimal between 0 and 1

To get a number between 1 and 10, multiply Math.random() \* 100 (or whatever you want upper limit to be

To get whole numbers and not fractions, use Math.floor() or Math.ceiling()

Math.floor() will give you 0 to 9 - for 1 through 10 you will have to add one - but for some reason this is how most tutorials say to do

Math.ceiling() will give you 1 to 10

* + - 1. floor/ceil
      2. cos/sin
      3. round
      4. pow
      5. abs
    1. Math constants
       1. PI
  1. JSON
     1. Definition
     2. Structure
     3. serialize/deserialize
     4. Methods
        1. stringify
        2. parse
  2. Date Object
     1. Creating
        1. Different ways to create
     2. How dates are stored
     3. Date methods
        1. toString/toUTCString/toDateString/toISOString
        2. getFullYear/getMonth/getDate/getHours/getMinutes/getSeconds/getMilliseconds/getTime/getDay
        3. Date.now
        4. UTC Date methods
        5. Set Date methods
           1. setDate
           2. setFullYear
           3. setHours
           4. setMilliseconds
           5. setMinutes
           6. setMonth
           7. setSeconds
           8. setTime
     4. ISO dates
     5. Time zones
     6. Long dates
     7. Parsing dates

1. Higher-order Functions
   1. Abstraction & repetition
   2. Higher-order functions definition
   3. Filtering arrays
   4. map
      1. Effect on original array
   5. reduce
      1. Array.reduce() lets you take an array of values, process each item in the array one at a time, and return a single final result
      2. Called reduce because it reduces the array down to one value
      3. Takes a callback function as an argument, which will be called one time for each item in the array
      4. Takes two arguments:
         1. previousResult, the value that the callback returned last time
         2. currentItem, the current item in the array
      5. the first time that the callback runs, there isn’t a previousResult available, so we need to also pass in an initial value that will be used as the first previousResult
   6. callback function doesn’t need to keep track of anything itself – it takes the previousResult and currentItem arguments, does something with them, and returns a new result value
   7. Flattening arrays
2. ‘this’
   1. In a method
   2. Alone
   3. In a function
      1. Default vs strict
   4. In event handlers
   5. Object method binding
   6. Explicit function binding
      1. call
      2. apply
      3. bind
   7. In arrow functions
3. More on objects
   1. Spread operator …
      1. Updating object immutably using spread operator
         1. return { …state, value: state.value + 1 } (from React)
   2. Encapsulation
   3. Prototypes
   4. Classes
      1. constructor
      2. using
   5. Class notation
   6. Overriding prototype properties
   7. Maps
   8. Polymorphism
   9. Iterators (next)
   10. Getters & setters
   11. Static
   12. Inheritance
       1. the problem - having multiple objects where we are repeating the same code   
          Example // Example is from Color Code - change for class  
          const me {  
           talk() {  
           return ‘Talking’;  
           }  
          }  
          const you {  
           talk() {  
           return ‘Talking’;  
           }  
          }
          1. Becomes problem if function is a lot of lines and needs to be in multiple objects
          2. Code bloat
          3. If error in one, may have to fix in all places
       2. Solution - use classes and inheritance   
          Example  
          class Person {  
           talk() {  
           return ‘Talking’;  
           }  
          }  
          const me = new Person();  
          const you = new Person();  
          me.talk(); // ‘Talking’  
          you.talk(); // ‘Talking’
          1. Function is in one place written one time
       3. If problem with talk function, how do we fix?
          1. It’s not on the me object:
             1. Putting me in the console shows:   
                Person {}  
                \_\_proto\_\_: Object
             2. Object with no methods
             3. If we type me.age = 12; and then put me in console, it shows:  
                Person { age: 12 }  
                age:12  
                \_\_proto\_\_: Object
          2. Talk function is on the prototype  
             Person {}  
             \_\_proto\_\_: Object  
             constructor: class Person  
             talk: f talk()  
             arguments: (…)  
             caller (…)  
             etc.  
             \_\_proto\_\_: Object
          3. If we want to update, we have to tap into prototype
          4. If we console log Person, it shows:  
             class Person {  
              talk() {  
              return ‘Talking’?  
              }  
             }
          5. But if we console.log Person.prototype, we see:  
             {constructor: f, talk: f}  
             constructor: class Person  
             talk: f talk()  
             \_\_proto\_\_: Object
          6. If we console.log Person.prototype === me.\_\_proto\_\_ we get true
             1. That means me.\_\_proto\_\_.talk is actually the talk function that’s on the prototype Person  
                result of console logging me.\_\_proto\_\_.talk:  
                f talk() {  
                 return ‘Talking’;  
                }
          7. Can fix talk function through Person prototype:  
             Person.prototype.talk = function()  
             {  
              return ‘New and improved talking’;  
             }  
             me.talk(); // New and improved talking  
             you.talk(): // New and improved talking
       4. Inheritance is actually done through objects and prototypes in JavaScript
          1. Called Prototypal Inheritance
   13. instanceOf
4. Bugs & errors
   1. console.log
   2. Dev tools
   3. Strict mode
      1. An optional way to force web browsers to run a restricted version of JavaScript
      2. Treats mistakes in your code that are normally considered bad style as actual errors that will cause your program not to run
      3. Makes your JavaScript code more secure and can make your programs run faster
      4. To enable for whole code:
         1. Put “use strict” before anything else in your JavaScript code
      5. To enable just for a function:
         1. Put “use strict” as the first statement within the function body  
            function myFunction() {  
             “use strict”;  
             // rest of the function here  
            }
      6. Some things that it does:
         1. Makes it impossible to accidentally create a global variable  
            myVariable = “hello World!”;  
            // non-strict mode – if myVariable doesn’t already exist, will create a global variable called myVariable  
            // strict mode – will cause an error
         2. Cause assignments that would normally just fail silently to throw errors  
            NaN = “a number!”;  
            // non-strict mode – trying to assign a value to a non-writable variable will just fail without alerting you  
            // strict mode – will cause an error
         3. Cause attempts to delete undeletable properties to cause an error  
            delete Object.prototype // throws an error in strict mode
         4. Requires that properties of an object are unique
         5. Requires that parameter names are unique
         6. Creates additional reserved keywords to prevent conflicts with keywords in future versions of JavaScript
         7. Source: https://www.dummies.com/article/technology/programming-web-design/javascript/javascript-strict-mode-144080/
   4. Types
   5. try/catch/finally/throw
   6. validation
   7. Testing
   8. Debugging
   9. Error propagation
   10. Exceptions
   11. Article to give to class: 10 Common JavaScript Bugs and How to Avoid Them - https://www.dummies.com/article/technology/programming-web-design/javascript/10-common-javascript-bugs-and-how-to-avoid-them-142585/
5. Regular expressions
   1. purpose
   2. Creating
   3. Testing for matches
   4. Sets of characters
      1. Modifiers
      2. Metacharacters
      3. quantifiers
   5. Repeating parts of a pattern
   6. Grouping subexpressions
   7. Matches and groups
   8. Date class
   9. Word and string boundaries
   10. Choice patterns
   11. How matching works
   12. Backtracking
   13. Replace method
   14. Greed
   15. Search method
   16. lastIndex property
6. Asynchronous programming
   1. Definition
   2. Callbacks - Allow us to wait on certain code to finish execution before running the next bit of code
   3. Promises
      1. Like callbacks, allow us to wait on certain code to finish execution before running the next bit of code
      2. Represents the eventual result of an asynchronous operation
      3. A placeholder - an object on which we can attach callbacks
      4. Promise states:
         1. Pending - async operation hasn’t completed yet
         2. Fulfilled - Operation has completed and the promise has a value
         3. Rejected - Operation has completed with an error or has failed
         4. A promise is settled if it is not pending
         5. Once a promise has settled, it is settled for good and cannot transition
      5. Promise syntax:
         1. runFunction().then(successFunc, failureFunc);
            1. First, invoke runFunction()
            2. runFunction() returns a Promise
      6. Once Promise is settled:
         1. If Promise is fulfilled, successFunc is invoked
         2. If Promise fails, failureFunc is. Invoked
   4. Event loop
      1. JavaScript executes code in an event-driven manner using the event loop.
      2. The message queue holds events to be processed
      3. example:  
         console.log("Start");  
         setTimeout(() => {  
          console.log("Inside Timeout");  
         }, 1000);

console.log("End");

* Even though setTimeout is set for 1 second, End gets logged first because the setTimeout callback goes to the message queue, and the event loop processes it after the stack is clear.
  + 1. setTimeout

1. DOM
   1. Trees
   2. Traversal
   3. Getting elements
   4. Changing the document
   5. Creating nodes
   6. Attributes
   7. Layout
   8. Styling
   9. querySelectors
2. Event handling
   1. Events and DOM nodes
   2. Event objects
   3. Propagation
   4. Default actions
   5. Types of events
      1. Key events
      2. Pointer events
      3. Scroll events
      4. Focus events
      5. Load event
   6. Event handler attributes
      1. Onclick
      2. Onchange
      3. Onmouseover
      4. Onmouseout
      5. Onkeydown
      6. onload
   7. Events and the event loop
   8. Timers
   9. Debouncing
3. Encapsulation
4. Prototypes
5. Classes
   1. constructor
   2. Using
   3. Class notation
   4. Overriding prototype properties
   5. Maps
   6. Polymorphism
   7. Iterators (next)
   8. Getters and setters
   9. Static
   10. Inheritance
   11. instanceOf
   12. Class vs Factory
       1. Factories are functions that generate and return objects
       2. classes provide a more organized and conventional way to structure objects, especially in scenarios involving inheritance
       3. Example

// Factory  
function createPerson(name, age) {  
 return {  
 name,  
 age,

greet() {  
 console.log(`Hello, my name is ${this.name}.`);  
 },  
 };  
}

// Class  
class Person {  
 constructor(name, age) {  
 this.name = name;  
 this.age = age;  
 }  
 greet() {  
 console.log(`Hello, my name is ${this.name}.`);  
 }  
}

const person1 = createPerson("John", 25);

const person2 = new Person("Alice", 30);

person1.greet(); // “Hello, my name is John."

person2.greet(); // "Hello, my name is Alice.”

1. Bugs & Errors
   1. Console.log
   2. Dev tools
   3. Strict Mode
   4. Types
   5. try/catch/finally/throw
   6. Validation
   7. Testing
   8. Debugging
   9. Error propagation
   10. exceptions
   11. subscribe() – returns an unsubscribe() function
2. Server-Side Programming
   1. Using MVC to Structure Our Application
   2. Node JS
      1. Introduction
      2. What is Node JS?
      3. Advantages of Node JS
         1. Traditional Web Server Model
         2. Node.js Process Model
      4. Setup Dev Environment
         1. Install Node.js on Windows
         2. Installing in mac os
      5. Working in REPL
      6. Node JS Console
         1. Creating package.json
         2. npm init
         3. Terminal will ask a series of questions
         4. To skip something, leave blank and hit enter
         5. To have terminal fill out everything for you:
         6. npm init -y
      7. Running a Node.js file through the CLI
         1. node <fieName>.js
      8. Referencing a module
         1. var header = require(‘../header.js);
      9. Launching the Node shell
         1. c:\>node  
            >console.log(“Hello World);
      10. Exiting the Node shell - Ctrl + D
      11. Stop server from running - Ctrl + C
      12. To build a Node application
          1. Create an application folder
          2. Create a package.json
             1. npm init
          3. Run npm install <moduleName> —save
   3. Modules for Node.js
      1. Express.js  
         npm install express —save  
         - do this from the folder/files working directory  
         var express = require(“express”) - requires express module  
         var app = express() - calls express function to start a new Express application  
         http.createServer(app).listen(8080) - starts server on port 8080
      2. Installing Express and EJS  
         npm install express ejs —save
   4. Modules
      1. Functions
      2. Buffer Module
      3. Module Types
         1. Core Modules
         2. Local Modules
      4. Module.Exports
   5. NPM
      1. What is NPM
      2. Installing Packages
         1. Locally
      3. Adding dependency in package.json
      4. Installing packages globally
      5. Updating packages
   6. Creating Web Server
      1. Handling http request
      2. Sending request
   7. Events
      1. EventEmitter class
      2. Returning event emitter
      3. Inhering events
   8. Express JS
      1. Configuring routes
      2. Working with express
      3. Express application structures - routes, views, and static files
      4. Serving Static Resources
      5. Serving static files
      6. Working with middle ware
3. Unit Seven - Debugging Node JS Application
   1. Core Node JS debugger
   2. Node Inspector
   3. Debugging with Visual Studio
      1. Set breakpoints to pause execution
      2. Debug variables
      3. Debug functions
   4. Unit Testing with Mocha
   5. Test Driven Development
      1. Automate and Organize Tests
      2. Write Expressive Tests
      3. Learn TDD with Mocha
4. Unit Eight - Database /RESTful Services
   1. Database Connectivity
   2. installing MySQL Workbench
   3. Create sample Database
   4. Connection string
   5. Configuring
   6. Working with select command
   7. RESTful Web Services
   8. Postman
   9. REST API creation and consumption using JSON format.
   10. Connection
   11. Collections
   12. Updating records
   13. Deleting records
5. Unit Nine - Template Engines
   1. Why template engine
   2. What is jade
   3. Simple tags
      1. Adding attributes to tags
      2. Blocks of text
      3. Powerful Features
      4. Loops
   4. Javascript
   5. Interpolation
   6. Mixins
   7. Putting it all together
   8. What is vash
   9. Configuration
   10. Using vash with express.js
   11. Template options
   12. Helpers
   13. API
   14. Example

Program Structure