JavaScript

* JS is the main scripting language of the web
* It’s used for providing dynamic client-side functionality and can be interpreted by all major browsers
* It’s a multi-paradigm language, meaning that we can use it for object-oriented programming, functional, or procedural programming
* It was created mainly for interacting and with manipulating the DOM
* There have frameworks such as Node.js that have been made to allow JS to be executed outside of a browser
* It’s a case-sensitive, loosely-typed, and dynamically-typed language
  + Loosely-typed languages don’t require us to declare the types of our variables when declaring the variables themselves
  + Dynamically-typed languages don’t determine the type of a variable until runtime and the type of a variable can change throughout execution with implicit conversion
* JS is an interpreted language, meaning that it can be read and executed one line at a time by our browsers - they don’t need a compiler to break down the JS into assembly for them to understand it

JavaScript Variable Data Types

* Primitive data types
  + String
    - Can be surrounded by either single or double quotes (or backticks to make use of template literals)
  + Boolean
  + Number
  + Undefined
    - The value that’s assigned to a variable if it’s not given a value when it’s declared
  + Symbol
  + Null
  + BigInt
    - Can hold integers of arbitrary length (number is a 64-bit data type)
* Any value that isn’t of a primitive data type is an object

Variable Declarations

* We don’t declare data types our variable declarations, but we do declare variables using one of three-scope enforcing keywords
  + var
    - enforces either global or function scope, depending on where we’ve declared the variable
    - if we declare a var variable outside of a function, it will have global scope and be able to be referenced from anywhere in our file
    - if we declare a var variable inside a function, it will have function scope and be able to be referenced from anywhere inside that function
  + let
    - enforces block scope, meaning that a let variable can be referenced within the block of code that declares it and any inner blocks
  + const
    - enforces block scope and issued for constant values
    - a const variable cannot have its changed after initial assignment\*
      * if we declare a const object, while the object that the variable holds can’t be changed (i.e. we can’t make the variable point to a different memory location), the values of the object’s properties can still be modified
  + if we’re working in strict mode, we won’t be able to declare a variable without using one of the three-scope enforcing keywords
  + var is an older way to declare variables and we should avoid it and use let or const instead
    - variables declared var can be redeclared later with var
    - variables that are declared with var are also hoisted
      * var variables are processed when the function starts execution (if they are function-scoped) or when the script starts execution (if they are globally-scoped)
      * var variables can be referenced before they’re declared
      * but when referenced before declaration, var variables will have an undefined value
      * this will be the case until execution reaches the code line where we declare the variable
  + a variable’s scope is the part of our code in which it can be referenced (i.e. the part of our code in which it’s reachable)

Truthy/Falsey Values

* falsey values are those that evaluate to false in a conditional expression or convert to false when used in a Boolean expression
* JS has six falsey values - false, 0, empty string (‘’ or “”), null, undefined, and NaN
* Every other value is truthy (including “0”)

Type Coercion

* Implicit conversion
  + Type conversion performed by the JavaScript engine without us explicitly converting a value from one data type to another
  + Numeric conversion
    - Occurs whenever we perform any mathematical operation (that doesn’t use the + operator) with at least one non-number value
    - If we attempt to divide using a value that isn’t a number, JavaScript will attempt to convert the value to a number

|  |  |
| --- | --- |
| Value | Becomes… |
| undefined | NaN |
| null | 0 |
| true | 1 |
| false | 0 |
| String | Whitespaces from start and end are removed. If the remaining is empty, the result 0. Otherwise, if the string can be converted to a number (i.e. the string only contains numeric values and at most a single decimal), its value will be that number. An error will result in NaN. |

* + String concatenation
    - If any of the values in an expression using the + operator is a string, the other will be converted to a string as well
  + Equality comparisons
    - When we use the equality operator (==) to compare two values of different data types, JavaScript will attempt to convert the values to numbers
    - We can use the strict equality operator (===) that will check the equality of the two values without performing any type conversion
* Explicit type conversion
  + Explicit type conversion occurs when we manually convert a value from one data type to another
  + We can do so by using, e.g. the String(), Number(), or Boolean() functions to convert a value to the respective primitive type
  + We can also use the double not operator (!!) to convert a value to its corresponding Boolean value

Template Literals

* We can surround strings by backticks (``) in place of single or double quotes
* Doing so allows us to create a template literal, where we can have multi-line string and we can embed expressions (including variables themselves) in our string by wrapping the expression ${}
* E.g.

let a = 9;

let b = `The current number is: ${a}`;

Control Flow

* Normally our code executes sequentially, one line at a time without repeating or skipping any lines
* Control flow allows us to change our execution path and repeat or skip lines of code (while only executing certain lines)
* Conditionals

if(conditionalExpression) {

} else if(conditionalExpression) {

} else {

}

* + Each conditional block must have one if statement, but it can have zero or more else if statements and zero or one else statements
  + The conditional expressions will be evaluated in order, the first conditional expression that’s true will have its associated code block executed and the remainder of the code blocks will be skipped
  + If none of the conditional expressions evaluate to true, the code block associated with the else statement (if present) will be executed
  + We can also use a ternary operator

conditionalExpression ? valueToReturnIfTrue : theValueToReturnIfFalse

* + - If the condition is true, the value between the ? and : will be returned
    - If the condition is false, the value after the : will be returned
* Switch statements

switch(expressionToEvaluate) {

case valueOne:

break;

case valueTwo:

break;

default:

}

* + Switch statements are good to use in place of repeated if-else if statements when we’re comparing the same expression to different values
  + If we don’t include a break statement at the end of a case, execution will fallthrough (i.e. it will continue and execute the code block(s) associated with the following case(s) until it reaches a break statement)
    - This fallthrough will occur if even the expression isn’t equivalent to the following values
    - We don’t have to include a break statement at the end of our default case
* While loops

while(conditionalExpression) {

}

* + The loop will continue to execute so long as the conditionalExpression evaluates to true
* Do-while loops

do {

} while(conditionalExpression);

* + The do-while loop will always execute at least once, regardless of the veracity of the conditionalExpression
  + The do-while loop will continue to execute so long as conditionalExpression remains true
* For loops

for(iterableVariableAssignment; exitCondition; incrementOrDecrementOperation) {

}

* + The for loop (or some form of it) is generally preferred over while loops because it’s more difficult to accidentally write an infinite loop (a loop that never finishes execution, such as a while loop where we write code such that our conditional expression never evaluates to false and we don’t include any sort of break statement)
* For-in loops

for(let iterableKey in objectToIterateThrough) {

}

* + Allows us to iterate through all keys in the object
* For-of loop

for(let iterableValue of iterableObject) {

}

* + Allows us to iterate over all the values of an iterable object (e.g. an array)

Arrays

* A type of object
* Zero-indexed lists of elements with comma-separated values
* The values within an array can have different data types
* We can initialize an array with the use of square brackets or by calling the array constructor, e.g.

let myArray = []; // empty array

let myArrayTwo = [1, ‘2’, 3];

let myArrayThree = new Array(1, 2, 3);

* + When using the array constructor, if we
    - Provide a single numeric value, we’ll get an array of the specified length and all elements will be undefined initially
    - Provide multiple values or a single, non-numeric argument, we’ll get an array with those values as its elements
* Because arrays are indexed, we can get the value of an element by its index by passing the index of the element in square brackets, e.g.

console.log(myArrayTwo[2]); // prints 3

* Arrays are dynamically-sized, meaning that the initial size of an array is not its fixed size (we can always add or remove elements later)
* Array methods/properties

|  |  |
| --- | --- |
| Method/Property | Explanation |
| shift() | Removes the first element of an array returns it |
| unshift() | Adds the specified element(s) to the beginning of the array |
| push() | Adds the specified element(s) to the end of the array |
| length | Holds the size of our array (i.e. the greatest index in the array incremented by one) |
| forEach() | Runs a function on each element of an array and mutate the array values |
| filter() | Returns an array of all array elements that match the filter |
| map() | Runs a function on each element of an array and returns an array of the results |
| reduce() | Runs a function on each element of an array, performing a calculation across array elements |

Objects

* Collections of key-value pairs
* We can declare them through an object literal, the Object.create(), etc., e.g.

let emptyObject = {};

let myObject = {keyOne: 1, keyTwo: 2};

* In an object, the key and associated value are colon-separated
* If we have multiple properties for our object (i.e. multiple key-value pairs), then we’ll comma-separate the properties
* We can access the values associated with our object’s keys through dot notation or square brackets, e.g.

console.log(myObject.keyOne);

console.log(myObject[keyTwo]);

* Object methods

|  |  |
| --- | --- |
| Method | Purpose |
| Object.keys() | Returns an array of all keys from the object passed as a parameter |
| Object.values() | Returns an array of all values from the object passed as a parameter |
| Object.entries() | Returns an array of key-value pairs from an object |

JSON

* JavaScript Object Notation
* Used both in and outside of JavaScript
* It’s a data format that’s commonly used to pass information between different web applications or services
* Different from JavaScript object literals in that properties and any strings that are values are always surrounded by double quotes
* The values for JSON can be object literals, arrays, or a value of the string, number, Boolean, or null primitives
  + i.e. the values can’t be functions, symbols, or undefined values
* JSON.stringify()
  + Takes an object as an argument and converts to a JSON string (aka a JSON-encoded object or serialized object)
* JSON.parse()
  + Takes a JSON string as an argument and converts it to a JavaScript object

JavaScript Operators

* Common JavaScript operators, listed in order from highest to lowest precedence
  + ()
    - Used for grouping
  + ++, --, -, +
    - ++ and -- are the postfix increment and decrement operators, respectively
      * Add/subtract one from a value, respectively
      * Because they’re postfixed, they follow the value they’re operating on (e.g. a++)
    - -, + are the unary negation and addition operators, respectively
      * Unary negation multiplies a value by -1
  + \*\*
    - Exponential operator
    - Allows us to take a value to the provided power (e.g. a \*\* 2, which gives us the square of a)
  + \*, /, %
    - % is the modulus operator
      * Its value is the remainder of a division operation (e.g. 4 % 3 is 1)
  + +, -
  + <, <=, >, >=, in
    - The in operator returns true if a given key is in an object and false otherwise
  + ==, !=, ===, !==
  + &&
    - Returns true so long as the values on both sides of the operator are true and false otherwise
  + ||
    - Returns true so long as one of the values on either side of the operator is true and false otherwise
  + =, +=, -=, \*\*=, \*=, /=, %=, &&=, ||=
    - Assignment operators
    - E.g.

let a = true;

a &&= 1 > 5; // false

JavaScript Comments

* Single line comments start with two forward slashes (//)
* Block or multiline comments begin with /\* and end with \*/

Console API

* The Console API contains methods that we can use in the browser to print output to our browser’s console
* console.log()
  + output the argument to the console
* console.error()
  + used to output an error to the console
* console.assert()
  + outputs an error message to the console if the assertion passed to is false

Functions

* a function is a block of code that runs when its invoked
* when declaring a function in JavaScript, we begin with the function keyword, then the name of the function, and any parameters that it takes in parentheses
  + when we declare function parameters, we only declare the parameter names themselves
* the body of function follow surrounded by curly braces
* e.g.

function nameOfFunction(parameterOne, parameterTwo) {

}

* functions in JavaScript are an object and a first-class citizen
  + this means that we can work with functions as if they were values of other data types (i.e. we can assign them to variables, as the values for keys in an object, etc.)
* function expressions
  + anonymous functions
  + function expressions are functions that are assigned to a variable or as the value for a key
  + to declare an anonymous function, we follow the function keyword with the function parameters in parentheses and the body of the function in curly braces
  + e.g.

let func = function(parameterOne, parameterTwo) {

};

* + - whenever we call func and pass it arguments for the parameters that the anonymous function assigned to it declares, that anonymous function will be executed
* arrow functions
  + shorthand anonymous functions
  + to write an arrow function, we have the parameters for the function in parentheses followed by the arrow operator (=>) and the function body in curly braces
  + e.g.

let func = (parameterOne, parameterTwo) => {

};

* + if our arrow function only declares one parameter, we can omit the parentheses
  + if the arrow function body is only a single line long, we can omit the curly braces

this

* a keyword that refers to the enclosing object
* we can use this keyword in any function, even if it’s not a method of an object
  + in this case, this will refer to the global object
* arrow functions don’t have their own context (i.e. their own this), so if we use this in an arrow function, then it will refer to the outer function/context

JavaScript Objects Continued

* toString()
  + if the toString() method for an object isn’t overridden, then it will return [object type]
    - type is the data type of the object
  + e.g. if we try to use the console.log() method to output the values of an object and we haven’t overridden the toString() method, then [object Object] will be output to the console
* accessor properties
  + functions that execute when getting and setting a value
  + these look like regular properties to external code
  + defined through the use of getter and setter methods
  + getter methods start with the get keyword, then have the name of the property
  + setter methods start with the set keyword, then the name of the property and take a single parameter (the value that will be assigned to the property)
  + we use accessor properties for encapsulation
    - whenever we want to allow external code to access the properties of our object, but control the ways in which it can do so (e.g. preventing a negative value from being assigned to a number variable)
  + e.g.

let obj = {

numberOfChickens: 0,

get propName() {

return numberOfChickens;

},

set propName(value) {

if(value >= 0) {

this.numberOfChickens = value;

}

}

}

* + accessor properties appear as regular properties to external code, so outside code could read from or assign to propName above by using dot notation or square brackets

Classes

* in object-oriented programming, a class is a blueprint from which objects will be instantiated
* to declare a class, we’ll start with the class keyword, followed by the name of the class, and the class body in curly braces
  + e.g.

class NameOfClass {

}

* to instantiate a class, we’ll use the new keyword to call the class constructor
  + e.g.

let a = new NameOfClass();

* classes in JavaScript always use strict mode
  + variables declared within a class are implicitly scoped to the class itself, so we don’t need to declare (and can’t) them with one of the three scope-enforcing keywords
  + but because the class uses strict mode, one the consequences of this is that we must declare any variables in our class functions with one of the scope-enforcing keywords
* classes can be defined through class expressions and assigned to variables
  + e.g.

let myClassVar = class {

}

* class constructors
  + constructors are functions that are executed when a class is instantiated (i.e. when a new object is made from that class)
  + constructors are used to perform setup for the new object, e.g. assigning initial values to the object’s properties
  + to create a constructor, we create a function in our class named constructor
  + constructors can take zero or more parameters
  + we can’t overload constructors for classes in JavaScript
  + e.g.

class MyClass {

constructor() {

}

}

* when we declare a function inside of a class or object, it’s referred to as a method
  + we don’t need to use the function keyword when declaring a method
* we can create static properties and methods for our classes by starting their declarations with the static keyword
  + the static keyword indicates that a method/property is associated with the class itself rather than an instance (i.e. object) of that class
  + therefore we can access the property or invoke the method without first instantiating the class

Class Inheritance

* we can have one class in JavaScript inherit from another by using the extends keyword in the class declaration
* e.g.

class ChildChild extends ParentClass {

}

* inheriting from a class allows us to use the parent class methods and properties (including those properties and methods that are static)
* we can directly inherit from at most one class
  + one class can extend no more than a single class, but we can have a chain of inheritance to where a child class inherits from a parent class, which itself inherits from a grandparent class
* constructors in children classes must call the constructor of the parent class with the super keyword and do so before they’re able to use the this keyword
  + e.g.

class ChildClass extends ParentClass {

classProperty;

constructor(valueForProperty) {

super();

this.classProperty = valueForProperty;

}

}

Private and Protected Fields

* a class variable that’s visible to outside code is called a property
* we can also refer to class variables as fields (whether they’re visible to outside code or not)
* when we declare a class variable, it’s available outside the class through objects of the class by default
* but we can make a variable private or protected to change this
* protected fields
  + those variables whose names begin with an \_
  + protected fields actually are accessible by outside entities, but by convention JS programmers treat them as protected (meaning that they should only be accessed from within the class that declares them or any children classes)
  + i.e. the language doesn’t actually enforce these restrictions on variables that start with \_ rather they’re just followed by convention
* private fields
  + those variables whose names begin with a #
  + they can’t be accessed outside of the class that declares them (there’s not an exception for inheritance - i.e. private fields can’t be accessed from children classes of the class declares them either)
  + private fields are actually enforced JavaScript, they’re not just a convention

JavaScript Modules

* a module is a JavaScript file that contains methods, variables, and/or classes that can be imported and used by another JavaScript file
* we use the export keyword to export variables, methods, and classes so that they can be used outside of the module where they’re declared
* we use the import keyword to import variables, methods, and classes so that we can use them
* modules are oftentimes used for code reusability (i.e. if we have a function that we want to invoke from multiple places, we can put in a module, export it, and have each file that needs it import it)
* modules (like classes) always use strict mode
  + variables that are defined within functions or in the module and outside of any other entity (like a class), must be declared with one of the scope-enforcing keywords
* exporting from modules
  + we can export a variable, method, or class from a module by using the export keyword
    - before its declaration
      * e.g.

export function add(numOne, numTwo) {

return numOne + numTwo;

}

* + - after the variable declaration and surround what we want to export in curly braces
      * e.g.

function add(numOne, numTwo) {

return numOne + numTwo;

}

export {add};

* + - to export multiple entities from a module in a single statement, we’ll just comma-separate them
  + exports that use the syntax we’ve described so far are named exports
  + but we can also create default exports by using the default keyword
    - if we’re using the export keyword when we declare the entity, default will be directly after export
      * e.g.

export default function add(numOne, numTwo) {

return numOne + numTwo;

}

* + - if we’re exporting something after we’ve declared it, we’ll just follow the name of the export with as default
      * e.g.

function add(numOne, numTwo) {

return numOne + numTwo;

}

export {add as default};

* + - there can only at most one default export per module
    - we aren’t required to provide a name for our default export
    - the default export is meant to be the main (or often times only) export from the module
* importing from modules
  + we can import a modules
    - named exports
      * by using an import statement as follows:

import {x, y} from “pathToModule.js”

* + - * we begin with import keyword followed by the entities that we want to import in curly braces
      * because we’re importing named exports, the names that we list in the curly braces must be the same as the names of the exports
      * our statement ends with the from keyword followed by a path to the module that exported what we’re importing
    - the default export
      * by using an import statement as follows:

import x from “pathToModule.js”

* + - * because we’re importing a default export, we omit the curly braces and we can name the import whatever we want to
    - both named and default exports
      * by using an import statement as follows:

import {default as x, y} from “pathToModule.js”

* + - * we indicate the variable that will hold the default export with the default as keywords
    - entire contents
      * by using an import statement as follows:

import \* as x from “pathToModule.js”

* + - * we use the asterisk to indicate that we want to import all of the exported variables, functions, and classes from the module
      * this is not preferred (we’d rather import the named or default exports that we want to work with explicitly) because we’ll need to use dot notation with the variable we declared anytime we want to use something we imported
  + a single file can import from multiple modules (we’ll just one import statement per module that we’re importing from)

The DOM

* the Document Object Model (DOM) is World Wide Web Consortium (W3C) standard for accessing documents
* in the HTML DOM, the Document is the html page/browser, the Objects are the elements in that page, and the Model is the way in which we access, interact with, change, add, and delete elements
* we can represent the DOM for an HTML page by an object tree with nodes for each object, where each object is an element on our page
* the root of the DOM tree for a particular page is the document (i.e. <html> element)
* the window object represents the browser and contains the entire document
* e.g.

Diagram

Description automatically generated

Accessing DOM Elements

* we can use the getElementById(), getElementsByClassName(), getElementsByTagName() methods on the document object in JavaScript to retrieve the matching children element(s)
  + getElementsByClassName() and getElementsByTagName() return arrays of matching elements
  + when calling one of these methods, we pass the value as the parameter to the method
  + e.g. to select the paragraph elements, we’d use document.getElementsByClassName(“contentSection”) or document.getElementsByTagName(“p”)
* we can also use the querySelector() method
  + takes CSS selector syntax as its argument and returns the first element that matches the selector
  + e.g. document.querySelector(“p.contentSection”)
* querySelectorAll()
  + takes CSS selector syntax as its argument and returns an array of elements that match the selector

[DOM Events](https://www.w3schools.com/jsref/dom_obj_event.asp)

[DOM Element Properties/Methods](https://www.w3schools.com/jsref/dom_obj_all.asp)

Events and Event Listeners

* events are actions that happen in the browser; they include such as user interaction (e.g. the click of a button) and browser-related actions (such as a page finishing loading)
* event listeners are functions that are added to HTML elements
  + once added to an element, the listener waits for an event to occur and then handles it when it does
* we can add event listeners on HTML elements in
  + HTML markup
    - we’ll assign the name of the event listener to the corresponding event attribute on the element
    - e.g.

<div onclick=”nameOfEventListener()”></div>

* + - * when the <div> element is clicked by the user, the nameOfEventListener() function will be invoked to handle it
  + JavaScript
    - With the addEventListener() method

Event Propagation

* Event propagation determines the order in which DOM elements are able to handle events once they occur
* There are two main types of propagation - bubbling and capturing
* When an event occurs, it goes through three phases
  + Capture phase
    - The event travels from the root of the DOM tree to the source element (i.e. the element where the event occurred, e.g. the <div> that was clicked)
  + Target phase
    - A bridge between the capture and bubble phases during which the event is at the source element
  + Bubble phase
    - The event travels from the source element back to the root of the DOM tree
* With event bubbling, event listeners act in the target and bubble phases
  + The source element can handle the event first (if it has an event listener for that event) and then any ancestor elements that have event listeners for that event may handle the event as it travels up to the root of the DOM tree
    - E.g.

Diagram

Description automatically generated

* With event capturing, event listeners act in the capture and target phases
  + The root element can handle the event first (if they have an event listener for that particular event) and then any ancestor elements with event listeners for that event may handle the event as it travels down to the source element
    - E.g.

Diagram

Description automatically generated

* I.e. events capture down and bubble up
* Bubbling is the more common form of propagation and the default method of event propagation

Adding and Removing Event Listeners in JavaScript

* addEventListener() and removeEventListener() methods add and remove an event listener from an element, respectively
  + each method takes three parameters
    - the first is type of the event to handle (e.g. “click” to handle the click event)
    - the second is the code to execute when the event occurs
    - the third is an optional Boolean value determining whether to handle the event through bubbling or capturing
      * if not specify, the value will be false, meaning that the event will be handled through bubbling
* we can pair these methods with the methods to retrieve elements from the DOM to add and remove event listeners in our JavaScript code

Event Object

* all events in JavaScript inherit from the Event object
  + therefore all events have common properties and methods inherited from the Event object
* stopPropagation() and stopImmediatePropagation()
  + both stop an event from continuing to travel through the DOM
  + if there are any listeners on the element whose listener called stopPropagation() that haven’t fired and act in the same phase, they’ll still be invoked
  + stopImmediatePropagation() prevents any further event listeners from firing, even if they’re on the same element whose listener called it
* cancelBubble()
  + used to stop the bubbling phase of an event
* cancellable and preventDefault()
  + cancellable property holds a Boolean value that indicates if we can cancel an event’s default action
  + preventDefault() method will cancel that default action (assuming that we can do so)
  + the default action of an event is not the same as the event propagation - it’s the action associated with HTML element when that event occurs (e.g. a redirection for an <a>)