JavaScript Review

Working with HTML, CSS, and JavaScript

While HTML and CSS provide website structure, JavaScript brings interactivity to websites by enabling complex functionality, such as handling user input, animating elements, and even building full web applications.

Data Types in JavaScript

Data types help the program understand the kind of data it's working with, whether it's a number, text, or something else.

- **Number**: A number represents both integers and floating-point values. Examples of integers include 7, 19, and 90.
- **Floating point**: A floating point number is a number with a decimal point. Examples include 3.14, 0.5, and 0.0001.
- **String**: A string is a sequence of characters, or text, enclosed in quotes. `"I like coding"` and `'JavaScript is fun'` are examples of strings.
- **Boolean**: A boolean represents one of two possible values: `true` or `false`. You can use a boolean to represent a condition, such as `isLoggedIn = true`.
- **Undefined and Null**: An undefined value is a variable that has been declared but not assigned a value. A null value is an empty value or a variable that has intentionally been assigned a value of `null`.
- **Object**: An object is a collection of key-value pairs. The key is the property name, and the value is the property value.

Here, the `pet` object has three properties or keys: `name`, `age`, and `type`. The values are `Fluffy`, `3`, and `dog`, respectively.

```
let pet = {
  name: 'Fluffy',
  age: 3,
  type: 'dog'
};
```

- **Symbol**: The Symbol data type is a unique and immutable value that may be used as an identifier for object properties.

In the example below, two symbols are created with the same description, but they are not equal.

```
const crypticKey1= Symbol('saltNpepper');
const crypticKey2= Symbol('saltNpepper');
console.log(crypticKey1 === crypticKey2); // false
```

- **BigInt**: When the number is too large for the `Number` data type, you can use the BigInt data type to represent integers of arbitrary length.

By adding an `n` to the end of the number, you can create a BigInt.

```
const veryBigNumber = 12345678901234567890123456789012345678901234567890n;
```

Variables in JavaScript

- Variables can be declared using the `let` keyword.

```
let cityName;
```

- To assign a value to a variable, you can use the assignment operator `=`.

```
cityName = 'New York';
```

- Variables declared using `let` can be reassigned a new value.

```
cityName = 'Los Angeles';
console.log(cityName); // Los Angeles
```

- Apart from `let`, you can also use `const` to declare a variable. However, a `const` variable cannot be reassigned a new value.

```
const cityName = 'New York';
cityName = 'Los Angeles'; // TypeError: Assignment to constant variable.
```

- Variables declared using `const` find uses in declaring constants, that are not allowed to change throughout the code, such as `PI` or `MAX_SIZE`.

Variable Naming Conventions

- Variable names should be descriptive and meaningful.
- Variable names should be camelCase like `cityName`, `isLoggedIn`, and `veryBigNumber`.
- Variable names should not start with a number. They must begin with a letter, `_`, or `$`.
- Variable names should not contain spaces or special characters, except for `_` and `$`.
- Variable names should not be reserved keywords.
- Variable names are case-sensitive. `age` and `Age` are different variables.

Strings and String immutability in JavaScript

- Strings are sequences of characters enclosed in quotes. They can be created using single quotes and double quotes.

```
let correctWay = 'This is a string';
let alsoCorrect = "This is also a string";
```

- Strings are immutable in JavaScript. This means that once a string is created, you cannot change the characters in the string. However, you can still reassign strings to a new value.

```
let firstName = 'John';
firstName = 'Jane'; // Reassigning the string to a new value
```

String Concatenation in JavaScript

- Concatenation is the process of joining multiple strings or combining strings with variables that hold text. The `+` operator is one of the simplest and most frequently used methods to concatenate strings.

```
let studentName = 'Asad';
let studentAge = 25;
let studentInfo = studentName + ' is ' + studentAge + ' years old.';
console.log(studentInfo); // Asad is 25 years old.
```

- If you need to add or append to an existing string, then you can use the `+=` operator. This is helpful when you want to build upon a string by adding more text to it over time.

```
let message = 'Welcome to programming, ';
message += 'Asad!';
console.log(message); // Welcome to programming, Asad!
```

- Another way you can concatenate strings is to use the `concat()` method. This method joins two or more strings together.

```
let firstName = 'John';
let lastName = 'Doe';
let fullName = firstName.concat(' ', lastName);
console.log(fullName); // John Doe
```

Logging Messages with `console.log()`

- The `console.log()` method is used to log messages to the console. It's a helpful tool for debugging and testing your code.

```
console.log('Hello, World!');
// Output: Hello, World!
```

Semicolons in JavaScript

- Semicolons are primarily used to mark the end of a statement. This helps the JavaScript engine understand the separation of individual instructions, which is crucial for correct execution.

```
let message = 'Hello, World!'; // first statement ends here
let number = 42; // second statement starts here
```

- Semicolons help prevent ambiguities in code execution and ensure that statements are correctly terminated.

Comments in JavaScript

- Any line of code that is commented out is ignored by the JavaScript engine. Comments are used to explain code, make notes, or temporarily disable code.
- Single-line comments are created using `//`.

```
// This is a single-line comment and will be ignored by the JavaScript engine
```

- Multi-line comments are created using `/*` to start the comment and `*/` to end the comment.

```
/*
This is a multi-line comment.
It can span multiple lines.
*/
```

JavaScript as a Dynamically Typed Language

- JavaScript is a dynamically typed language, which means that you don't have to specify the data type of a variable when you declare it. The JavaScript engine automatically determines the data type based on the value assigned to the variable.

```
let error = 404; // JavaScript treats error as a number
error = "Not Found"; // JavaScript now treats error as a string
```

- Other languages, like Java, that are not dynamically typed would result in an error:

```
int error = 404; // value must always be an integer
error = "Not Found"; // This would cause an error in Java
```

Using the `typeof` Operator

- The `typeof` operator is used to check the data type of a variable. It returns a string indicating the type of the variable.

```
let age = 25;
console.log(typeof age); // number

let isLoggedIn = true;
console.log(typeof isLoggedIn); // boolean
```

- However, there's a well-known quirk in JavaScript when it comes to null. The `typeof` operator returns `object` for null values.

```
let user = null;
console.log(typeof user); // object
```

String Basics

- **Definition**: A string is a sequence of characters wrapped in either single quotes, double quotes or backticks. Strings are primitive data types and they are immutable. Immutability means that once a string is created, it cannot be changed.
- **Accessing Characters from a String**: To access a character from a string you can use bracket notation and pass in the index number. An index is the position of a character within a string, and it is zero-based.

```
const developer = "Jessica";
developer[0] // J
```

- `\n` **(Newline Character)**: You can create a newline in a string by using the `\n` newline character.

```
const poem = "Roses are red,\nViolets are blue,\nJavaScript is fun,\nAnd so are you.";
console.log(poem);
```

- **Escaping Strings**: You can escape characters in a string by placing backlashes (`\`) in front of the quotes.

```
const statement = "She said, \"Hello!\"";
console.log(statement); // She said, "Hello!"
```

Template Literals (Template Strings) and String Interpolation

- **Definition**: Template literals are defined with backticks (`). They allow for easier string manipulation, including embedding variables directly inside a string, a feature known as string interpolation.

```
const name = "Jessica";
const greeting = `Hello, ${name}!`; // "Hello, Jessica!"
```

ASCII, the `charCodeAt()` Method and the `fromCharCode()` Method

- **ASCII**: ASCII, short for American Standard Code for Information Interchange, is a character encoding standard used in computers to represent text. It assigns a numeric value to each character, which is universally recognized by machines.
- **The `charCodeAt()` Method**: This method is called on a string and returns the ASCII code of the character at a specified index.

```
const letter = "A";
console.log(letter.charCodeAt(0));  // 65
```

- **The `fromCharCode()` Method**: This method converts an ASCII code into its corresponding character.

```
const char = String.fromCharCode(65);
console.log(char);  // A
```

Other Common String Methods

- **The `indexOf` Method**: This method is used to search for a substring within a string. If the substring is found, `indexOf` returns the index (or position) of the first occurrence of that substring. If the substring is not found, `indexOf` returns -1, which indicates that the search was unsuccessful.

```
const text = "The quick brown fox jumps over the lazy dog.";
console.log(text.indexOf("fox")); // 16
console.log(text.indexOf("cat")); // -1
```

- **The `includes()` Method**: This method is used to check if a string contains a specific substring. If the substring is found within the string, the method returns true. Otherwise, it returns false.

```
const text = "The quick brown fox jumps over the lazy dog.";
console.log(text.includes("fox")); // true
console.log(text.includes("cat")); // false
```

- The `slice()` **Method**: This method returns a new array containing a shallow copy of a portion of the original array, specified by start and end indices. The new array contains references to the same elements as the original array (not duplicates). This means that if the elements are primitives (like numbers or strings), the values are copied; but if the elements are objects or arrays, the references are copied, not the objects themselves.

```
const text = "freeCodeCamp";
console.log(text.slice(0, 4));  // "free"
console.log(text.slice(4, 8));  // "Code"
console.log(text.slice(8, 12)); // "Camp"
```

- **The** `toUpperCase()` **Method**: This method converts all the characters to uppercase letters and returns a new string with all uppercase characters.

```
const text = "Hello, world!";
console.log(text.toUpperCase()); // "HELLO, WORLD!"
```

- **The** `toLowerCase()` **Method**: This method converts all characters in a string to lowercase.

```
const text = "HELLO, WORLD!"
console.log(text.toLowerCase()); // "hello, world!"
```

- **The** `replace()` **Method**: This method allows you to find a specified value (like a word or character) in a string and replace it with another value. The method returns a new string with the replacement and leaves the original unchanged because JavaScript strings are immutable.

```
const text = "I like cats";
console.log(text.replace("cats", "dogs")); // "I like dogs"
```

- **The** `repeat()` **Method**: This method is used to repeat a string a specified number of times.

```
const text = "Hello";
console.log(text.repeat(3)); // "HelloHelloHello"
```

- **The** `trim()` **Method**: This method is used to remove whitespaces from both the beginning and the end of a string.

```
const text = "  Hello, world!  ";
console.log(text.trim()); // "Hello, world!"
```

- **The** `trimStart()` **Method**: This method removes whitespaces from the beginning (or "start") of the string.

```
const text = "  Hello, world!  ";
console.log(text.trimStart()); // "Hello, world!  "
```

- **The `trimEnd()` Method**: This method removes whitespaces from the end of the string.

```
const text = " Hello, world! ";
console.log(text.trimEnd()); // "  Hello, world!"
```

- **The `prompt()` Method**: This method of the `window` is used to get information from a user through the form of a dialog box. This method takes two arguments. The first argument is the message which will appear inside the dialog box, typically prompting the user to enter information. The second one is a default value which is optional and will fill the input field initially.

```
const answer = window.prompt("What's your favorite animal?"); // This will change depending on what the user answers
```

Working with the Number Data Type

- **Definition**: JavaScript's `Number` type includes integers, floating-point numbers, `Infinity` and `NaN`. Floating-point numbers are numbers with a decimal point. Positive `Infinity` is a number greater than any other number while `-Infinity` is a number smaller than any other number. `NaN` (`Not a Number`) represents an invalid numeric value like the string `"Jessica"`.

Common Arithmetic Operations

- **Addition Operator**: This operator (`+`) is used to calculate the sum of two or more numbers.
- **Subtraction Operator**: This operator (`-`) is used to calculate the difference between two numbers.
- **Multiplication Operator**: This operator (`*`) is used to calculate the product of two or more numbers.
- **Division Operator**: This operator (`/`) is used to calculate the quotient between two numbers
- **Division By Zero**: If you try to divide by zero, JavaScript will return `Infinity`.
- **Remainder Operator**: This operator(`%`) returns the remainder of a division.
- **Exponentiation Operator**: This operator (`**`) raises one number to the power of another.

Calculations with Numbers and Strings

- **Explanation**: When you use the `+` operator with a number and a string, JavaScript will coerce the number into a string and concatenate the two values. When you use the `-`, `*` or `/` operators with a string and number, JavaScript will coerce the string into a number and the result will be a number. For `null` and `undefined`, JavaScript treats `null` as 0 and undefined as `NaN` in mathematical operations.

```
const result = 5 + '10';

console.log(result); // 510
console.log(typeof result); // string

const subtractionResult = '10' - 5;
console.log(subtractionResult); // 5
console.log(typeof subtractionResult); // number
```

```
const multiplicationResult = '10' * 2;
console.log(multiplicationResult); // 20
console.log(typeof multiplicationResult); // number

const divisionResult = '20' / 2;
console.log(divisionResult); // 10
console.log(typeof divisionResult); // number

const result1 = null + 5;
console.log(result1); // 5
console.log(typeof result1); // number

const result2 = undefined + 5;
console.log(result2); // NaN
console.log(typeof result2); // number
```

Operator Precedence

- **Definition**: Operator precedence determines the order in which operations are evaluated in an expression. Operators with higher precedence are evaluated before those with lower precedence. Values inside the parenthesis will be evaluated first and multiplication/division will have higher precedence than addition/subtraction. If the operators have the same precedence, then JavaScript will use associativity. Associativity is what tells JavaScript whether to evaluate operators from left to right or right to left. For example, the exponent operator is also right to left associative:

```
const result = (2 + 3) * 4;

console.log(result); // 20

const result2 = 10 - 2 + 3;

console.log(result2); // 11

const result3 = 2 ** 3 ** 2;

console.log(result3); // 512
```

Increment and Decrement Operators

- **Increment Operator**: This operator is used to increase the value by one. The prefix notation `++num` increases the value of the variable first, then returns a new value. The postfix notation `num++` returns the current value of the variable first, then increases it.

```
let x = 5;
```

```
console.log(++x); // 6
console.log(x); // 6


let y = 5;

console.log(y++); // 5
console.log(y); // 6
```

- **Decrement Operator**: This operator is used to decrease the value by one. The prefix and postfix notation works the same way as earlier with the increment operator.

```
let num = 5;

console.log(--num); // 4
console.log(num--); // 4
console.log(num); // 3
```

Compound Assignment Operators

- **Addition Assignment (`+=`) Operator**: This operator performs addition on the values and assigns the result to the variable.
- **Subtraction Assignment (`-=`) Operator**: This operator performs subtraction on the values and assigns the result to the variable.
- **Multiplication Assignment (`*=`) Operator**: This operator performs multiplication on the values and assigns the result to the variable.
- **Division Assignment (`/=`) Operator**: This operator performs division on the values and assigns the result to the variable.
- **Remainder Assignment (`%=`) Operator**: This operator divides a variable by the specified number and assigns the remainder to the variable.
- **Exponentiation Assignment (`**=`) Operator**: This operator raises a variable to the power of the specified number and reassigns the result to the variable.

Booleans and Equality

- **Boolean Definition**: A boolean is a data type that can only have two values: `true` or `false`.
- **Equality (`==`) Operator**: This operator uses type coercion before checking if the values are equal.

```
console.log(5 == '5'); // true
```

- **Strict Equality (`===`) Operator**: This operator does not perform type coercion and checks if both the types and values are equal.

```
console.log(5 === '5'); // false
```

- **Inequality (`!=`) Operator**: This operator uses type coercion before checking if the values are not equal.
- **Strict Inequality (`!==`) Operator**: This operator does not perform type coercion and checks if both the types and values are not equal.

Comparison Operators

- **Greater Than (`>`) Operator**: This operator checks if the value on the left is greater than the one on the right.
- **Greater Than (`>=`) or Equal Operator**: This operator checks if the value on the left is greater than or equal to the one on the right.

- **Less Than (`<`) Operator**: This operator checks if the value on the left is less than the one on the right.
- **Less Than (`<=`) or Equal Operator**: This operator checks if the value on the left is less than or equal to the one on the right.

Unary Operators

- **Unary Plus Operator**: This operator converts its operand into a number. If the operand is already a number, it remains unchanged.

```
const str = '42';
const num = +str;

console.log(num); // 42
console.log(typeof num); // number
```

- **Unary Negation (`-`) Operator**: This operator negates the operand.

```
const num = 4;
console.log(-num); // -4
```

- **Logical NOT (`!`) Operator**: This operator flips the boolean value of its operand. So, if the operand is `true`, it becomes `false`, and if it's `false`, it becomes `true`.

Bitwise Operators

- **Bitwise AND (`&`) Operator**: This operator returns a 1 in each bit position for which the corresponding bits of both operands are 1.
- **Bitwise AND Assignment (`&=`) Operator**: This operator performs a `bitwise AND` operation with the specified number and reassigns the result to the variable.
- **Bitwise OR (`|`) Operator**: This operator returns a 1 in each bit position for which the corresponding bits of either or both operands are 1.
- **Bitwise OR Assignment (`|=`) Operator**: This operator performs a `bitwise OR` operation with the specified number and reassigns the result to the variable.
- **Bitwise XOR (`^`) Operator**: This operator returns a 1 in each bit position for which the corresponding bits of either, but not both, operands are 1.
- **Bitwise NOT (`~`) Operator**: This operator inverts the binary representation of a number.
- **Left Shift (`<<`) Operator**: This operator shifts all bits to the left by a specified number of positions.
- **Right Shift (`>>`) Operator**: This operator shifts all bits to the right.

Conditional Statements, Truthy Values, Falsy Values and the Ternary Operator

- **`if/else if/else`**: An `if` statement takes a condition and runs a block of code if that condition is `truthy`. If the condition is `false`, then it moves to the `else if` block. If none of those conditions are `true`, then it will execute the `else` clause. `Truthy` values are any values that result in `true` when evaluated in a Boolean context like an `if` statement. `Falsy` values are values that evaluate to `false` in a Boolean context.

```
const score = 87;

if (score >= 90) {
 console.log('You got an A');
} else if (score >= 80) {
 console.log('You got a B'); // You got an B
} else if (score >= 70) {
 console.log('You got a C');
```

```
} else {
 console.log('You failed! You need to study more!');
}
```

- **Ternary Operator**: This operator is often used as a shorter way to write `if else` statements.

```
const temperature = 30;
const weather = temperature > 25 ? 'sunny' : 'cool';

console.log(`It's a ${weather} day!`); // It's a sunny day!
```

Binary Logical Operators

- **Logical AND (`&&`) Operator**: This operator checks if both operands are truthy. If the first value is truthy, then it will return the second value. If the first value is falsy, then it will return the first value.

```
const result = true && 'hello';

console.log(result); // hello
```

- **Logical OR (`||`) Operator**: This operator checks if at least one of the operands is truthy. If the first value is truthy, then it is returned. If the first value is falsy, then the second value is returned.
- **Nullish Coalescing (`??`) Operator**: This operator will return a value only if the first one is `null` or `undefined`.

```
const userSettings = {
 theme: null,
 volume: 0,
 notifications: false,
};

let theme = userSettings.theme ?? 'light';
console.log(theme); // light
```

The `Math` Object

- **The `Math.random()` Method**: This method generates a random floating-point number between 0 (inclusive) and 1 (exclusive). This means the possible output can be 0, but it will never actually reach 1.
- **The `Math.max()` Method**: This method takes a set of numbers and returns the maximum value.
- **The `Math.min()` Method**: This method takes a set of numbers and returns the minimum value.
- **The `Math.ceil()` Method**: This method rounds a value up to the nearest whole integer.
- **The `Math.floor()` Method**: This method rounds a value down to the nearest whole integer.
- **The `Math.round()` Method**: This method rounds a value to the nearest whole integer.

```
console.log(Math.round(2.3)); // 2
console.log(Math.round(4.5)); // 5
console.log(Math.round(4.8)); // 5
```

- **The `Math.trunc()` Method**: This method removes the decimal part of a number, returning only the integer portion, without rounding.
- **The `Math.sqrt()` Method**: This method will return the square root of a number.
- **The `Math.cbrt()` Method**: This method will return the cube root of a number.
- **The `Math.abs()` Method**: This method will return the absolute value of a number.
- **The `Math.pow()` Method**: This method takes two numbers and raises the first to the power of the second.

Common Number Methods

- **`isNaN()`**: `NaN` stands for "Not-a-Number". It's a special value that represents an unrepresentable or undefined numerical result. The `isNaN()` function property is used to determine whether a value is `NaN` or not. `Number.isNaN()` provides a more reliable way to check for `NaN` values, especially in cases where type coercion might lead to unexpected results with the global `isNaN()` function.

```
console.log(isNaN(NaN));       // true
console.log(isNaN(undefined)); // true
console.log(isNaN({}));        // true

console.log(isNaN(true));      // false
console.log(isNaN(null));      // false
console.log(isNaN(37));        // false


console.log(Number.isNaN(NaN));        // true
console.log(Number.isNaN(Number.NaN)); // true
console.log(Number.isNaN(0 / 0));      // true

console.log(Number.isNaN("NaN"));      // false
console.log(Number.isNaN(undefined));  // false
```

- **The `parseFloat()` Method**: This method parses a string argument and returns a floating-point number. It's designed to extract a number from the beginning of a string, even if the string contains non-numeric characters later on.
- **The `parseInt()` Method**: This method parses a string argument and returns an integer. `parseInt()` stops parsing at the first non-digit it encounters. For floating-point numbers, it returns only the integer part. If it can't find a valid integer at the start of the string, it returns `NaN`.
- **The `toFixed()` Method**: This method is called on a number and takes one optional argument, which is the number of digits to appear after the decimal point. It returns a string representation of the number with the specified number of decimal places.

Comparisons and the `null` and `undefined` Data Types

- **Comparisons and `undefined`**: A variable is `undefined` when it has been declared but hasn't been assigned a value. It's the default value of uninitialized variables and function parameters that weren't provided an argument. `undefined` converts to `NaN` in numeric contexts, which makes all numeric comparisons with `undefined` return `false`.

```
console.log(undefined > 0);  // false
console.log(undefined < 0);  // false
console.log(undefined == 0); // false
```

- **Comparisons and `null`**: The `null` type represents the intentional absence of a value. When using the equality operator, `null` and `undefined` are considered equal. However, when using the strict equality operator (`===`), which checks both value and type without performing type coercion, `null` and `undefined` are not equal:

```
console.log(null == undefined); // true
console.log(null === undefined); // false
```

`switch` Statements

- **Definition**: A `switch` statement evaluates an expression and matches its value against a series of `case` clauses. When a match is found, the code block associated with that case is executed.

```
const dayOfWeek = 3;

switch (dayOfWeek) {
  case 1:
    console.log("It's Monday! Time to start the week strong.");
    break;
  case 2:
    console.log("It's Tuesday! Keep the momentum going.");
    break;
  case 3:
    console.log("It's Wednesday! We're halfway there.");
    break;
  case 4:
    console.log("It's Thursday! Almost the weekend.");
    break;
  case 5:
    console.log("It's Friday! The weekend is near.");
    break;
  case 6:
    console.log("It's Saturday! Enjoy your weekend.");
    break;
  case 7:
    console.log("It's Sunday! Rest and recharge.");
    break;
  default:
```

```
    console.log("Invalid day! Please enter a number between 1 and 7.");
}
```

JavaScript Functions

- Functions are reusable blocks of code that perform a specific task.
- Functions can be defined using the `function` keyword followed by a name, a list of parameters, and a block of code that performs the task.
- Arguments are values passed to a function when it is called.
- When a function finishes its execution, it will always return a value.
- By default, the return value of a function is `undefined`.
- The `return` keyword is used to specify the value to be returned from the function and ends the function execution.

Arrow Functions

- Arrow functions are a more concise way to write functions in JavaScript.
- Arrow functions are defined using the `=>` syntax between the parameters and the function body.
- When defining an arrow function, you do not need the `function` keyword.
- If you are using a single parameter, you can omit the parentheses around the parameter list.
- If the function body consists of a single expression, you can omit the curly braces and the `return` keyword.

Scope in Programming

- **Global scope**: This is the outermost scope in JavaScript. Variables declared in the global scope are accessible from anywhere in the code and are called global variables.
- **Local scope**: This refers to variables declared within a function. These variables are only accessible within the function where they are declared and are called local variables.
- **Block scope**: A block is a set of statements enclosed in curly braces `{}` such as in `if` statements, or loops.
- Block scoping with `let` and `const` provides even finer control over variable accessibility, helping to prevent errors and make your code more predictable.

JavaScript Array Basics

- **Definition**: A JavaScript array is an ordered collection of values, each identified by a numeric index. The values in a JavaScript array can be of different data types, including numbers, strings, booleans, objects, and even other arrays. Arrays are contiguous in memory, which means that all elements are stored in a single, continuous block of memory locations, allowing for efficient indexing and fast access to elements by their index.

```
const developers = ["Jessica", "Naomi", "Tom"];
```

- **Accessing Elements From Arrays**: To access elements from an array, you will need to reference the array followed by its index number inside square brackets. JavaScript arrays are zero based indexed which means the first element is at index 0, the second element is at index 1, etc. If you try to access an index that doesn't exist for the array, then JavaScript will return `undefined`.

```
const developers = ["Jessica", "Naomi", "Tom"];
developers[0] // "Jessica"
developers[1] // "Naomi"

developers[10] // undefined
```

- `length` **Property**: This property is used to return the number of items in an array.

```
const developers = ["Jessica", "Naomi", "Tom"];
developers.length // 3
```

- **Updating Elements in an Array**: To update an element in an array, you use the assignment operator (`=`) to assign a new value to the element at a specific index.

```
const fruits = ['apple', 'banana', 'cherry'];
fruits[1] = 'blueberry';

console.log(fruits); // ['apple', 'blueberry', 'cherry']
```

Two Dimensional Arrays

- **Definition**: A two-dimensional array is essentially an array of arrays. It's used to represent data that has a natural grid-like structure, such as a chessboard, a spreadsheet, or pixels in an image. To access an element in a two-dimensional array, you need two indices: one for the row and one for the column.

```
const chessboard = [
    ['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R'],
    ['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    ['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'],
    ['r', 'n', 'b', 'q', 'k', 'b', 'n', 'r']
];

console.log(chessboard[0][3]); // "Q"
```

Array Destructuring

- **Definition**: Array destructuring is a feature in JavaScript that allows you to extract values from arrays and assign them to variables in a more concise and readable way. It provides a convenient syntax for unpacking array elements into distinct variables.

```
const fruits = ["apple", "banana", "orange"];

const [first, second, third] = fruits;

console.log(first); // "apple"
console.log(second); // "banana"
console.log(third); // "orange"
```

- **Rest Syntax**: This allows you to capture the remaining elements of an array that haven't been destructured into a new array.

```
const fruits = ["apple", "banana", "orange", "mango", "kiwi"];
const [first, second, ...rest] = fruits;

console.log(first); // "apple"
console.log(second); // "banana"
console.log(rest); // ["orange", "mango", "kiwi"]
```

Common Array Methods

- `push()` **Method**: This method is used to add elements to the end of the array and will return the new length.

```
const desserts = ["cake", "cookies", "pie"];
desserts.push("ice cream");

console.log(desserts); // ["cake", "cookies", "pie", "ice cream"];
```

- `pop()` **Method**: This method is used to remove the last element from an array and will return that removed element. If the array is empty, then the return value will be `undefined`.

```
const desserts = ["cake", "cookies", "pie"];
desserts.pop();

console.log(desserts); // ["cake", "cookies"];
```

- `shift()` **Method**: This method is used to remove the first element from an array and return that removed element. If the array is empty, then the return value will be `undefined`.

```
const desserts = ["cake", "cookies", "pie"];
desserts.shift();

console.log(desserts); // ["cookies", "pie"];
```

- `unshift()` **Method**: This method is used to add elements to the beginning of the array and will return the new length.

```
const desserts = ["cake", "cookies", "pie"];
desserts.unshift("ice cream");

console.log(desserts); // ["ice cream", "cake", "cookies", "pie"];
```

- `indexOf()` **Method**: This method is useful for finding the first index of a specific element within an array. If the element cannot be found, then it will return `-1`.

```
const fruits = ["apple", "banana", "orange", "banana"];
const index = fruits.indexOf("banana");
```

```
console.log(index); // 1
console.log(fruits.indexOf("not found")); // -1
```

- `splice()` **Method**: This method is used to add or remove elements from any position in an array. The return value for the `splice()` method will be an array of the items removed from the array. If nothing was removed, then an empty array will be returned. This method will mutate the original array, modifying it in place rather than creating a new array. The first argument specifies the index at which to begin modifying the array. The second argument are the number of elements you wish to remove. The following arguments are the elements you wish to add.

```
const colors = ["red", "green", "blue"];
colors.splice(1, 0, "yellow", "purple");

console.log(colors); // ["red", "yellow", "purple", "green", "blue"]
```

- `includes()` **Method**: This method is used to check if an array contains a specific value. This method returns `true` if the array contains the specified element, and `false` otherwise.

```
const programmingLanguages = ["JavaScript", "Python", "C++"];

console.log(programmingLanguages.includes("Python")); // true
console.log(programmingLanguages.includes("Perl")); // false
```

- `concat()` **Method**: This method creates a new array by merging two or more arrays.

```
const programmingLanguages = ["JavaScript", "Python", "C++"];
const newList = programmingLanguages.concat("Perl");

console.log(newList); // ["JavaScript", "Python", "C++", "Perl"]
```

- `slice()` **Method**: This method returns a shallow copy of a portion of the array, starting from a specified index or the entire array. A shallow copy will copy the reference to the array instead of duplicating it.

```
const programmingLanguages = ["JavaScript", "Python", "C++"];
const newList = programmingLanguages.slice(1);

console.log(newList); // ["Python", "C++"]
```

- **Spread Syntax**: The spread syntax is used to create shallow copies of an array.

```
const originalArray = [1, 2, 3];
const shallowCopiedArray = [...originalArray];
```

```
shallowCopiedArray.push(4);

console.log(originalArray); // [1, 2, 3]
console.log(shallowCopiedArray); // [1, 2, 3, 4]
```

- `split()` **Method**: This method divides a string into an array of substrings and specifies where each split should happen based on a given separator. If no separator is provided, the method returns an array containing the original string as a single element.

```
const str = "hello";
const charArray = str.split("");

console.log(charArray); // ["h", "e", "l", "l", "o"]
```

- `reverse()` **Method**: This method reverses an array in place.

```
const desserts = ["cake", "cookies", "pie"];
console.log(desserts.reverse()); // ["pie", "cookies", "cake"]
```

- `join()` **Method**: This method concatenates all the elements of an array into a single string, with each element separated by a specified separator. If no separator is provided, or an empty string (`""`) is used, the elements will be joined without any separator.

```
const reversedArray = ["o", "l", "l", "e", "h"];
const reversedString = reversedArray.join("");

console.log(reversedString); // "olleh"
```

Object Basics

- **Definition**: An object is a data structure that is made up of properties. A property consists of a key and a value. To access data from an object you can use either dot notation or bracket notation.

```
const person = {
  name: "Alice",
  age: 30,
  city: "New York"
};

console.log(person.name);  // Alice
console.log(person["name"]); // Alice
```

To set a property of an existing object you can use either dot notation or bracket notation together with the assignment operator.

```
const person = {
  name: "Alice",
  age: 30
};

person.job = "Engineer"
person["hobby"] = "Knitting"
console.log(person);  // {name: 'Alice', age: 30, job: 'Engineer', hobby: 'Knitting'}
```

Removing Properties From an Object

- `delete` **Operator**: This operator is used to remove a property from an object.

```
const person = {
  name: "Alice",
  age: 30,
  job: "Engineer"
};

delete person.job;

console.log(person.job); // undefined
```

Checking if an Object has a Property

- `hasOwnProperty()` **Method**: This method returns a boolean indicating whether the object has the specified property as its own property.

```
const person = {
  name: "Alice",
  age: 30
};

console.log(person.hasOwnProperty("name")); // true
console.log(person.hasOwnProperty("job")); // false
```

- `in` **Operator**: This operator will return `true` if the property exists in the object.

```
const person = {
  name: "Bob",
  age: 25
};
```

```
console.log("name" in person);  // true
```

Accessing Properties From Nested Objects

- **Accessing Data**: Accessing properties from nested objects involves using the dot notation or bracket notation, much like accessing properties from simple objects. However, you'll need to chain these accessors to drill down into the nested structure.

```
const person = {
  name: "Alice",
  age: 30,
  contact: {
    email: "alice@example.com",
    phone: {
      home: "123-456-7890",
      work: "098-765-4321"
    }
  }
};

console.log(person.contact.phone.work); // "098-765-4321"
```

Primitive and Non Primitive Data Types

- **Primitive Data Types**: These data types include numbers, strings, booleans, `null`, `undefined`, and symbols. These types are called "primitive" because they represent single values and are not objects. Primitive values are immutable, which means once they are created, their value cannot be changed.
- **Non Primitive Data Types**: In JavaScript, these are objects, which include regular objects, arrays, and functions. Unlike primitives, non-primitive types can hold multiple values as properties or elements.

Object Methods

- **Definition**: Object methods are functions that are associated with an object. They are defined as properties of an object and can access and manipulate the object's data. The `this` keyword inside the method refers to the object itself, enabling access to its properties.

```
const person = {
  name: "Bob",
  age: 30,
  sayHello: function() {
    return "Hello, my name is " + this.name;
  }
};

console.log(person.sayHello()); // "Hello, my name is Bob"
```

Object Constructor

- **Definition**: In JavaScript, a constructor is a special type of function used to create and initialize objects. It is invoked with the `new` keyword and can initialize properties and methods on the newly created object. The `Object()` constructor creates a new empty object.

```
new Object()
```

Working with the Optional Chaining Operator (`?.`)

- **Definition**: This operator lets you safely access object properties or call methods without worrying whether they exist.

```
const user = {
  name: "John",
  profile: {
    email: "john@example.com",
    address: {
      street: "123 Main St",
      city: "Somewhere"
    }
  }
};

console.log(user.profile?.address?.street); // "123 Main St"
console.log(user.profile?.phone?.number);   // undefined
```

Object Destructuring

- **Definition**: Object destructuring allows you to extract values from objects and assign them to variables in a more concise and readable way.

```
const person = { name: "Alice", age: 30, city: "New York" };

const { name, age } = person;

console.log(name); // Alice
console.log(age);  // 30
```

Working with JSON

- **Definition**: JSON stands for JavaScript Object Notation. It is a lightweight, text-based data format that is commonly used to exchange data between a server and a web application.

```
{
  "name": "Alice",
  "age": 30,
```

```
  "isStudent": false,
  "list of courses": ["Mathematics", "Physics", "Computer Science"]
}
```

- `JSON.stringify()`: This method is used to convert a JavaScript object into a JSON string. This is useful when you want to store or transmit data in a format that can be easily shared or transferred between systems.

```
const user = {
  name: "John",
  age: 30,
  isAdmin: true
};

const jsonString = JSON.stringify(user);
console.log(jsonString); // '{"name":"John","age":30,"isAdmin":true}'
```

- `JSON.parse()`: This method converts a JSON string back into a JavaScript object. This is useful when you retrieve JSON data from a web server or localStorage and you need to manipulate the data in your application.

```
const jsonString = '{"name":"John","age":30,"isAdmin":true}';
const userObject = JSON.parse(jsonString);

// result: { name: 'John', age: 30, isAdmin: true }
console.log(userObject);
```

Working with Loops

- `for` **Loop**: This type of loop is used to repeat a block of code a certain number of times. This loop is broken up into three parts: the initialization statement, the condition, and the increment/decrement statement. The initialization statement is executed before the loop starts. It is typically used to initialize a counter variable. The condition is evaluated before each iteration of the loop. An iteration is a single pass through the loop. If the condition is `true`, the code block inside the loop is executed. If the condition is `false`, the loop stops and you move on to the next block of code. The increment/decrement statement is executed after each iteration of the loop. It is typically used to increment or decrement the counter variable.

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

- `for...of` **Loop**: This type of loop is used when you need to loop over values from an iterable. Examples of iterables are arrays and strings.

```
const numbers = [1, 2, 3, 4, 5];

for (const num of numbers) {
```

```
  console.log(num);
}
```

- **`for...in` Loop**: This type of loop is best used when you need to loop over the properties of an object. This loop will iterate over all enumerable properties of an object, including inherited properties and non-numeric properties.

```
const fruit = {
  name: 'apple',
  color: 'red',
  price: 0.99
};

for (const prop in fruit) {
  console.log(fruit[prop]);
}
```

- **`while` Loop**: This type of loop will run a block of code as long as the condition is `true`.

```
let i = 5;

while (i > 0) {
  console.log(i);
  i--;
}
```

- **`do...while` Loop**: This type of loop will execute the block of code at least once before checking the condition.

```
let userInput;

do {
  userInput = prompt("Please enter a number between 1 and 10");
} while (Number(userInput) < 1 || Number(userInput) > 10);

alert("You entered a valid number!");
```

`break` and `continue` Statements

- **Definition**: A `break` statement is used to exit a loop early, while a `continue` statement is used to skip the current iteration of a loop and move to the next one.

```
// Example of break statement
for (let i = 0; i < 10; i++) {
  if (i === 5) {
```

```
    break;
  }
  console.log(i);
}

// Output: 0, 1, 2, 3, and 4

// Example of continue statement
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    continue;
  }
  console.log(i);
}

// Output: 0, 1, 2, 3, 4, 6, 7, 8, and 9
```

String Constructor and `toString()` Method

- **Definition**: A string object is used to represent a sequence of characters. String objects are created using the `String` constructor function, which wraps the primitive value in an object.

```
const greetingObject = new String("Hello, world!");

console.log(typeof greetingObject); // "object"
```

- `toString()` **Method**: This method converts a value to its string representation. It is a method you can use for numbers, booleans, arrays, and objects.

```
const num = 10;
console.log(num.toString()); // "10"

const arr = [1, 2, 3];
console.log(arr.toString()); // "1,2,3"
```

This method accepts an optional radix which is a number from 2 to 36. This radix represents the base, such as base 2 for binary or base 8 for octal. If the radix is not specified, it defaults to base 10, which is decimal.

```
const num = 10;
console.log(num.toString(2)); // "1010"(binary)
```

Number Constructor

- **Definition**: The `Number` constructor is used to create a number object. The number object contains a few helpful properties and methods like the `isNaN` and `toFixed` method. Most of the time, you will be using the `Number` constructor to convert other data types to the number data type.

```
const myNum = new Number("34");
console.log(typeof myNum); // "object"

const num = Number('100');
console.log(num); // 100

console.log(typeof num); // number
```

Best Practices for Naming Variables and Functions

- **camelCasing**: By convention, JavaScript developers will use camel casing for naming variables and functions. Camel casing is where the first word is all lowercase and the following words start with a capital letter. Ex. `isLoading`.
- **Naming Booleans**: For boolean variables, it's a common practice to use prefixes such as "is", "has", or "can".

```
let isLoading = true;
let hasPermission = false;
let canEdit = true;
```

- **Naming Functions**: For functions, the name should clearly indicate what the function does. For functions that return a boolean (often called predicates), you can use the same "is", "has", or "can" prefixes. When you have functions that retrieve data, it is common to start with the word "get". When you have functions that set data, it is common to start with the word "set". For event handler functions, you might prefix with "handle" or suffix with "Handler".

```
function getUserData() { /* ... */ }

function isValidEmail(email) { /* ... */ }

function getProductDetails(productId) { /* ... */ }

function setUserPreferences(preferences) { /* ... */ }

function handleClick() { /* ... */ }
```

- **Naming Variables Inside Loops**: When naming iterator variables in loops, it's common to use single letters like `i`, `j`, or `k`.

```
for (let i = 0; i < array.length; i++) { /* ... */ }
```

Working with Sparse Arrays

- **Definition**: It is possible to have arrays with empty slots. Empty slots are defined as slots with nothing in them. This is different than array slots with the value of `undefined`. These types of arrays are known as sparse arrays.

```
const sparseArray = [1, , , 4];
console.log(sparseArray.length); // 4
```

Linters and Formatters

- **Linters**: A linter is a static code analysis tool that flags programming errors, bugs, stylistic errors, and suspicious constructs. An example of a common linter would be ESLint.
- **Formatters**: Formatters are tools that automatically format your code to adhere to a specific style guide. An example of a common formatter is Prettier.

Memory Management

- **Definition**: Memory management is the process of controlling the memory, allocating it when needed and freeing it up when it's no longer needed. JavaScript uses automatic memory management. This means that JavaScript (more specifically, the JavaScript engine in your web browser) takes care of memory allocation and deallocation for you. You don't have to explicitly free up memory in your code. This automatic process is often called "garbage collection."

Closures

- **Definition**: A closure is a function that has access to variables in its outer (enclosing) lexical scope, even after the outer function has returned.

```
function outerFunction(x) {
  let y = 10;
  function innerFunction() {
    console.log(x + y);
  }
  return innerFunction;
}

let closure = outerFunction(5);
closure(); // 15
```

`var` Keyword and Hoisting

- **Definition**: `var` was the original way to declare variables before 2015. But there were some issues that came with `var` in terms of scope, redeclaration and more. So that is why modern JavaScript programming uses `let` and `const` instead.
- **Redeclaring Variables with `var`**: If you try to redeclare a variable using `let`, then you would get a `SyntaxError`. But with `var`, you are allowed to redeclare a variable.

```
// Uncaught SyntaxError: Identifier 'num' has already been declared
let num = 19;
let num = 18;

var myNum = 5;
var myNum = 10; // This is allowed and doesn't throw an error

console.log(myNum) // 10
```

- **`var` and Scope**: Variables declared with `var` inside a block (like an `if` statement or a `for` loop) are still accessible outside that block.

```
if (true) {
  var num = 5;
}
console.log(num); // 5
```

- **Hoisting**: This is JavaScript's default behavior of moving declarations to the top of their respective scopes during the compilation phase before the code is executed. When you declare a variable using the `var` keyword, JavaScript hoists the declaration to the top of its scope.

```
console.log(num); // undefined
var num = 5;
console.log(num); // 5
```

When you declare a function using the function declaration syntax, both the function name and the function body are hoisted. This means you can call a function before you've declared it in your code.

```
sayHello(); // "Hello, World!"

function sayHello() {
  console.log("Hello, World!");
}
```

Variable declarations made with `let` or `const` are hoisted, but they are not initialized, and you can't access them before the actual declaration in your code. This behavior is often referred to as the "temporal dead zone".

```
console.log(num); // Throws a ReferenceError
let num = 10;
```

Working with Imports, Exports and Modules

- **Module**: This is a self-contained unit of code that encapsulates related functions, classes, or variables. To create a module, you write your JavaScript code in a separate file.
- **Exports**: Any variables, functions, or classes you want to make available to other parts of your application need to be explicitly exported using the `export` keyword. There are two types of export: named export and default export.
- **Imports**: To use the exported items in another part of your application, you need to import them using the `import` keyword. The types can be named import, default import, and namespace import.

```
// Within a file called math.js, we export the following functions:

// Named export
export function add(num1, num2) {
```

```
  return num1 + num2;
}

// Default export
export default function subtract(num1, num2) {
  return num1 - num2;
}

// Within another file, we can import the functions from math.js.

// Named import - This line imports the add function.
// The name of the function must exactly match the one exported from math.js.
import { add } from './math.js';

// Default import - This line imports the subtract function.
// The name of the function can be anything.
import subtractFunc from './math.js';

// Namespace import - This line imports everything from the file.
import * as Math from './math.js';

console.log(add(5, 3)); // 8
console.log(subtractFunc(5, 3)); // 2
console.log(Math.add(5, 3)); // 8
console.log(Math.subtract(5, 3)); // 2
```

Callback Functions and the `forEach` Method

- **Definition**: In JavaScript, a callback function is a function that is passed as an argument to another function and is executed after the main function has finished its execution.
- **`forEach()` Method**: This method is used to iterate over each element in an array and perform an operation on each element. The callback function in `forEach` can take up to three arguments: the current element, the index of the current element, and the array that `forEach` was called upon.

```
const numbers = [1, 2, 3, 4, 5];

// Result: 2 4 6 8 10
numbers.forEach((number) => {
  console.log(number * 2);
});
```

Higher Order Functions

- **Definition**: A higher-order function takes one or more functions for the arguments and returns a function or value for the result.

```
function operateOnArray(arr, operation) {
  const result = [];
  for (let i = 0; i < arr.length; i++) {
    result.push(operation(arr[i]));
  }
  return result;
}

function double(x) {
  return x * 2;
}

const numbers = [1, 2, 3, 4, 5];
const doubledNumbers = operateOnArray(numbers, double);
console.log(doubledNumbers); // [2, 4, 6, 8, 10]
```

- `map()` **Method**: This method is used to create a new array by applying a given function to each element of the original array. The callback function can accept up to three arguments: the current element, the index of the current element, and the array that `map` was called upon.

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((num) => num * 2);

console.log(numbers); // [1, 2, 3, 4, 5]
console.log(doubled); // [2, 4, 6, 8, 10]
```

- `filter()` **Method**: This method is used to create a new array with elements that pass a specified test, making it useful for selectively extracting items based on criteria. Just like the `map` method, the callback function for the `filter` method accepts the same three arguments: the current element being processed, the index, and the array.

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const evenNumbers = numbers.filter((num) => num % 2 === 0);

console.log(evenNumbers); // [2, 4, 6, 8, 10]
```

- `reduce()` **Method**: This method is used to process an array and condense it into a single value. This single value can be a number, a string, an object, or even another array. The `reduce()` method works by applying a function to each element in the array, in order, passing the result of each calculation on to the next. This function is often called the reducer function. The reducer function takes two main parameters: an accumulator and the current value. The accumulator is where you store the running result of your operations, and the current value is the array element being processed.

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce(
  (accumulator, currentValue) => accumulator + currentValue,
```

```
  0
);

console.log(sum); // 15
```

Method Chaining

- **Definition**: Method chaining is a programming technique that allows you to call multiple methods on the same object in a single line of code. This technique can make your code more readable and concise, especially when performing a series of operations on the same object.

```
const result = "  Hello, World!  "
  .trim()
  .toLowerCase()
  .replace("world", "JavaScript");

console.log(result); // "hello, JavaScript!"
```

Working with the `sort` Method

- **Definition**: The `sort` method is used to sort the elements of an array and return a reference to the sorted array. No copy is made in this case because the elements are sorted in place.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();

console.log(fruits); // ["Apple", "Banana", "Mango", "Orange"]
```

If you need to sort numbers, then you will need to pass in a compare function. The `sort` method converts the elements to strings and then compares their sequences of UTF-16 code units values. UTF-16 code units are the numeric values that represent the characters in the string. Examples of UTF-16 code units are the numbers 65, 66, and 67 which represent the characters "A", "B", and "C" respectively. So the number 200 appears before the number 3 in an array, because the string "200" comes before the string "3" when comparing their UTF-16 code units.

```
const numbers = [414, 200, 5, 10, 3];

numbers.sort((a, b) => a - b);

console.log(numbers); // [3, 5, 10, 200, 414]
```

The parameters `a` and `b` are the two elements being compared. The compare function should return a negative value if `a` should come before `b`, a positive value if `a` should come after `b`, and zero if `a` and `b` are equal.

Working with the `every` and `some` Methods

- `every()` **Method**: This method tests whether all elements in an array pass a test implemented by a provided function. The `every()` method returns `true` if the provided function returns `true` for all elements in the array. If any element fails the test, the method immediately returns `false` and stops checking the remaining elements.

```
const numbers = [2, 4, 6, 8, 10];
const hasAllEvenNumbers = numbers.every((num) => num % 2 === 0);

console.log(hasAllEvenNumbers); // true
```

- `some()` **Method**: This method checks if at least one element passes the test. The `some()` method returns `true` as soon as it finds an element that passes the test. If no elements pass the test, it returns `false`.

```
const numbers = [1, 3, 5, 7, 8, 9];
const hasSomeEvenNumbers = numbers.some((num) => num % 2 === 0);

console.log(hasSomeEvenNumbers); // true
```

Working with the DOM and Web APIs

- **API**: An API (Application Programming Interface) is a set of rules and protocols that allow software applications to communicate with each other and exchange data efficiently.
- **Web API**: Web APIs are specifically designed for web applications. These types of APIs are often divided into two main categories: browser APIs and third-party APIs.
- **Browser APIs**: These APIs expose data from the browser. As a web developer, you can access and manipulate this data using JavaScript.
- **Third-Party APIs**: These are not built into the browser by default. You have to retrieve their code in some way. Usually, they will have detailed documentation explaining how to use their services. An example is the Google Maps API, which you can use to display interactive maps on your website.
- **DOM**: The DOM stands for Document Object Model. It's a programming interface that lets you interact with HTML documents. With the DOM, you can add, modify, or delete elements on a webpage. The root of the DOM tree is the `html` element. It's the top-level container for all the content of an HTML document. All other nodes are descendants of this root node. Then, below the root node, we find other nodes in the hierarchy. A parent node is an element that contains other elements. A child node is an element that is contained within another element.
- `navigator` **Interface**: This provides information about the browser environment, such as the user agent string, the platform, and the version of the browser. A user agent string is a text string that identifies the browser and operating system being used.
- `window` **Interface**: This represents the browser window that contains the DOM document. It provides methods and properties for interacting with the browser window, such as resizing the window, opening new windows, and navigating to different URLs.

Working with the `querySelector()`, `querySelectorAll()` and `getElementById()` Methods

- `getElementById()` **Method**: This method is used to get an object that represents the HTML element with the specified `id`. Remember that IDs must be unique in every HTML document, so this method will only return one Element object.

```
<div id="container"></div>
```

```
const container = document.getElementById("container");
```

- `querySelector()` **Method**: This method is used to get the first element in the HTML document that matches the CSS selector passed as an argument.

```
<section class="section"></section>
```

```
const section = document.querySelector(".section");
```

- `querySelectorAll()` **Method**: You can use this method to get a list of all the DOM elements that match a specific CSS selector.

```
<ul class="ingredients">
  <li>Sugar</li>
  <li>Milk</li>
  <li>Eggs</li>
</ul>
```

```
const ingredients = document.querySelectorAll('ul.ingredients li');
```

Working with the `innerText()`, `innerHTML()`, `createElement()` and `textContent()` Methods

- `innerHTML` **Property**: This is a property of the `Element` that is used to set or update parts of the HTML markup.

```
<div id="container">
  <!-- Add new elements here -->
</div>
```

```
const container = document.getElementById("container");
container.innerHTML = '<ul><li>Cheese</li><li>Tomato</li></ul>';
```

- `createElement` **Method**: This is used to create an HTML element.

```
const img = document.createElement("img");
```

- `innerText`: This represents the visible text content of the HTML element and its descendants.

```
<div id="container">
  <p>Hello, World!</p>
  <p>I'm learning JavaScript</p>
</div>
```

```
const container = document.getElementById("container");
console.log(container.innerText);
```

- `textContent`: This returns the plain text content of an element, including all the text within its descendants.

```
<div id="container">
  <p>Hello, World!</p>
  <p>I'm learning JavaScript</p>
</div>
```

```
const container = document.getElementById("container");
console.log(container.textContent);
```

Working with the `appendChild()` and `removeChild()` Methods

- `appendChild()` **Method**: This method is used to add a node to the end of the list of children of a specified parent node.

```
<ul id="desserts">
  <li>Cake</li>
  <li>Pie</li>
</ul>
```

```
const dessertsList = document.getElementById("desserts");
const listItem = document.createElement("li");

listItem.textContent = "Cookies";
dessertsList.appendChild(listItem);
```

- `removeChild()` **Method**: This method is used to remove a node from the DOM.

```
<section id="example-section">
  <h2>Example sub heading</h2>
  <p>first paragraph</p>
  <p>second paragraph</p>
</section>
```

```
const sectionEl = document.getElementById("example-section");
const lastParagraph = document.querySelector("#example-section p:last-of-type");

sectionEl.removeChild(lastParagraph);
```

Work with the `setAttribute` Method

- **Definition**: This method is used to set the attribute for a given element. If the attribute already exists, then the value is updated. Otherwise, a new attribute is added with a value.

```
<p id="para">I am a paragraph</p>
```

```
const para = document.getElementById("para");
para.setAttribute("class", "my-class");
```

Event Object

- **Definition**: The `Event` object is a payload that triggers when a user interacts with your web page in some way. These interactions can be anything from clicking on a button or focusing an input to shaking their mobile device. All `Event` objects will have the `type` property. This property reveals the type of event that triggered the payload, such as keydown or click. These values will correspond to the same values you might pass to `addEventListener()`, where you can capture and utilize the `Event` object.

`addEventListener()` and `removeEventListener()` Methods

- `addEventListener` **Method**: This method is used to listen for events. It takes two arguments: the event you want to listen for and a function that will be called when the event occurs. Some common examples of events would be click events, input events, and change events.

```
const btn = document.getElementById("btn");

btn.addEventListener("click", () => alert("You clicked the button"));
```

- `removeEventListener` **Method**: This method is used to remove an event listener that was previously added to an element using the `addEventListener` method. This is useful when you want to stop listening for a particular event on an element.

```
const bodyEl = document.querySelector("body");
const para = document.getElementById("para");
const btn = document.getElementById("btn");

let isBgColorGrey = true;

function toggleBgColor() {
  bodyEl.style.backgroundColor = isBgColorGrey ? "blue" : "grey";
  isBgColorGrey = !isBgColorGrey;
}

btn.addEventListener("click", toggleBgColor);

para.addEventListener("mouseover", () => {
  btn.removeEventListener("click", toggleBgColor);
});
```

- **Inline Event Handlers**: Inline event handlers are special attributes on an HTML element that are used to execute JavaScript code when an event occurs. In modern JavaScript, inline event handlers are not considered best practice. It is preferred to use the `addEventListener` method instead.

```
<button onclick="alert('Hello World!')">Show alert</button>
```

DOMContentLoaded

- **Definition**: The `DOMContentLoaded` event is fired when everything in the HTML document has been loaded and parsed. If you have external stylesheets or images, the `DOMContentLoaded` event will not wait for those to be loaded. It will only wait for the HTML to be loaded.

Working with `style` and `classList`

- `Element.style` **Property**: This property is a read-only property that represents the inline style of an element. You can use this property to get or set the style of an element.

```
const paraEl = document.getElementById("para");
paraEl.style.color = "red";
```

- `Element.classList` **Property**: This property is a read-only property that can be used to add, remove, or toggle classes on an element.

```
// Example adding a class
const paraEl = document.getElementById("para");
paraEl.classList.add("highlight");

// Example removing a class
paraEl.classList.remove("blue-background");

// Example toggling a class
const menu = document.getElementById("menu");
const toggleBtn = document.getElementById("toggle-btn");

toggleBtn.addEventListener("click", () => menu.classList.toggle("show"));
```

Working with the `setTimeout` and `setInterval` Methods

- `setTimeout()` **Method**: This method lets you delay an action for a specified time.

```
setTimeout(() => {
 console.log('This runs after 3 seconds');
}, 3000);
```

- `setInterval()` **Method**: This method keeps running a piece of code repeatedly at a set interval. Since `setInterval()` keeps executing the provided function at the specified interval, you might want to stop it. For this, you have to use the `clearInterval()` method.

```
setInterval(() => {
 console.log('This runs every 2 seconds');
}, 2000);

// Example using clearInterval
const intervalID = setInterval(() => {
 console.log('This will stop after 5 seconds');
}, 1000);

setTimeout(() => {
 clearInterval(intervalID);
}, 5000);
```

The `requestAnimationFrame()` Method

- **Definition**: This method allows you to schedule the next step of your animation before the next screen repaint, resulting in a fluid and visually appealing experience. The next screen repaint refers to the moment when the browser refreshes the visual display of the web page. This happens multiple times per second, typically around 60 times (or 60 frames per second) on most displays.

```
function animate() {
 // Update the animation...
 // for example, move an element, change a style, and more.
 update();
 // Request the next frame
 requestAnimationFrame(animate);
}
```

Web Animations API

- **Definition**: The Web Animations API lets you create and control animations directly inside JavaScript.

```
const square = document.querySelector('#square');

const animation = square.animate(
 [{ transform: 'translateX(0px)' }, { transform: 'translateX(100px)' }],
 {
   duration: 2000, // makes animation lasts 2 seconds
   iterations: Infinity, // loops indefinitely
   direction: 'alternate', // moves back and forth
   easing: 'ease-in-out', // smooth easing
```

```
  }
);
```

The Canvas API

- **Definition**: The Canvas API is a powerful tool that lets you manipulate graphics right inside your JavaScript file. To work with the Canvas API, you first need to provide a `canvas` element in HTML. This element acts as a drawing surface you can manipulate with the instance methods and properties of the interfaces in the Canvas API. This API has interfaces like `HTMLCanvasElement`, `CanvasRenderingContext2D`, `CanvasGradient`, `CanvasPattern`, and `TextMetrics` which contain methods and properties you can use to create graphics in your JavaScript file.

```
<canvas id="my-canvas" width="400" height="400"></canvas>
```

```
const canvas = document.getElementById('my-canvas');

// Access the drawing context of the canvas.
// "2d" allows you to draw in two dimensions
const ctx = canvas.getContext('2d');

// Set the background color
ctx.fillStyle = 'crimson';

// Draw a rectangle
ctx.fillRect(1, 1, 150, 100);
```

Opening and Closing Dialogs and Modals with JavaScript

- **Modal and Dialog Definitions**: Dialogs let you display important information or actions to users. With the HTML built-in dialog element, you can easily create these dialogs (both modal and non-modal dialogs) in your web apps. A modal dialog is a type of dialog that forces the user to interact with it before they can access the rest of the application or webpage. In contrast, a non-modal dialog allows the user to continue interacting with other parts of the page or application even when the dialog is open. It doesn't prevent access to the rest of the content.
- `showModal()` **Method**: This method is used to open a modal.

```
<dialog id="my-modal">
   <p>This is a modal dialog.</p>
</dialog>
<button id="open-modal">Open Modal Dialog</button>
```

```
const dialog = document.getElementById('my-modal');
const openButton = document.getElementById('open-modal');

openButton.addEventListener('click', () => {
```

```
  dialog.showModal();
});
```

- `close()` **Method**: This method is used to close the modal.

```
<dialog id="my-modal">
   <p>This is a modal dialog.</p>
   <button id="close-modal">Close Modal</button>
</dialog>
<button id="open-modal">Open Modal Dialog</button>
```

```
const dialog = document.getElementById('my-modal');
const openButton = document.getElementById('open-modal');
const closeButton = document.getElementById('close-modal');

openButton.addEventListener('click', () => {
  dialog.show();
});

closeButton.addEventListener('click', () => {
  dialog.close();
});
```

The Change Event

- **Definition**: The change event is a special event which is fired when the user modifies the value of certain input elements. Examples would include when a checkbox or a radio button is ticked. Or when the user makes a selection from something like a date picker or dropdown menu.

```
<label>
  Choose a programming language:
  <select class="language" name="language">
    <option value="">---Select One---</option>
    <option value="JavaScript">JavaScript</option>
    <option value="Python">Python</option>
    <option value="C++">C++</option>
  </select>
</label>

<p class="result"></p>
```

```
const selectEl = document.querySelector(".language");
const result = document.querySelector(".result");

selectEl.addEventListener("change", (e) => {
  result.textContent = `You enjoy programming in ${e.target.value}.`;
});
```

Event Bubbling

- **Definition**: Event bubbling, or propagation, refers to how an event "bubbles up" to parent objects when triggered.

Event Delegation

- **Definition**: Event delegation is the process of listening to events that have bubbled up to a parent, rather than handling them directly on the element that triggered them.

JavaScript and Accessibility

Common ARIA Accessibility Attributes

- `aria-expanded` **attribute**: Used to convey the state of a toggle (or disclosure) feature to screen reader users.
- `aria-haspopup` **attribute**: This state is used to indicate that an interactive element will trigger a pop-up element when activated. You can only use the `aria-haspopup` attribute when the pop-up has one of the following roles: `menu`, `listbox`, `tree`, `grid`, or `dialog`. The value of `aria-haspopup` must be either one of these roles or `true`, which is the same as `menu`.
- `aria-checked` **attribute**: This attribute is used to indicate whether an element is in the checked state. It is most commonly used when creating custom checkboxes, radio buttons, switches, and listboxes.
- `aria-disabled` **attribute**: This state is used to indicate that an element is disabled only to people using assistive technologies, such as screen readers.
- `aria-selected` **attribute**: This state is used to indicate that an element is selected. You can use this state on custom controls like a tabbed interface, a listbox, or a grid.
- `aria-controls` **attribute**: Used to associate an element with another element that it controls. This helps people using assistive technologies understand the relationship between the elements.
- `hidden` **attribute**: Hides inactive panels from both visual and assistive technology users.

Working with Live Regions and Dynamic Content

- `aria-live` **attribute**: Makes part of a webpage a live region, meaning any updates inside that area will be announced by a screen reader so users don't miss important changes.
- `polite` **value**: Most live regions use this value. This value means that the update is not urgent, so the screen reader can wait until it finishes any current announcement or the user completes their current action before announcing the update.

Here is an example of a live region that is dynamically updated by JavaScript:

```
<div aria-live="polite" id="status"></div>
```

```
const statusEl = document.getElementById("status");
statusEl.textContent = "Your file has been successfully uploaded.";
```

- `contenteditable` **attribute**: Turns the element into a live editor, allowing users to update its content as if it were a text field. When there is no visible label or heading for a contenteditable region, add an accessible name using the `aria-label` attribute to help screen reader users understand the purpose of the editable area.

```
<div contenteditable="true" aria-label="Note editor">
  Editable content goes here
</div>
```

`focus` and `blur` Events

- `blur` **event**: Fires when an element loses focus.

```
element.addEventListener("blur", () => {
  // Handle when user leaves the element
});
```

- `focus` **event**: Fires when an element receives focus.

```
element.addEventListener("focus", () => {
  // Handle when user enters the element
});
```

Common Types of Error Messages

- **SyntaxError**: These errors happen when you write something incorrectly in your code, like missing a parenthesis, or a bracket. Think of it like a grammar mistake in a sentence.

```
const arr = ["Beau", "Quincy" "Tom"]
```

- **ReferenceError**: There are several types of Reference Errors, triggered in different ways. The first type of reference error would be not defined variables. Another example of a ReferenceError is trying to access a variable, declared with `let` or `const`, before it has been defined.

```
console.log(num);
const num = 50;
```

- **TypeError**: These errors occur when you try to perform an operation on the wrong type.

```
const developerObj = {
  name: "Jessica",
  country: "USA",
  isEmployed: true
};

developerObj.map()
```

- **RangeError**: These errors happen when your code tries to use a value that's outside the range of what JavaScript can handle.

```
const arr = [];
arr.length = -1;
```

The `throw` Statement

- **Definition**: The `throw` statement in JavaScript is used to throw a user-defined exception. An exception in programming, is when an unexpected event happens and disrupts the normal flow of the program.

```
function validateNumber(input) {
  if (typeof input !== "number") {
    throw new TypeError("Expected a number, but received " + typeof input);
  }
  return input * 2;
}
```

`try...catch...finally`

- **Definition**: The `try` block is used to wrap code that might throw an error. It acts as a safe space to try something that could fail. The `catch` block captures and handles errors that occur in the try block. You can use the error object inside catch to inspect what went wrong. The `finally` block runs after the try and catch blocks, regardless of whether an error occurred. It's commonly used for cleanup tasks, such as closing files or releasing resources.

```
function processInput(input) {
  if (typeof input !== "string") {
    throw new TypeError("Input must be a string.");
  }

  return input.toUpperCase();
}

try {
  console.log("Starting to process input...");
  const result = processInput(9);
  console.log("Processed result:", result);
} catch (error) {
  console.error("Error occurred:", error.message);
}
```

Debugging Techniques

- `debugger` **Statement**: This statement lets you pause your code at a specific line to investigate what's going on in the program.

```
let firstNumber = 5;
let secondNumber = 10;

debugger; // Code execution pauses here
let sum = firstNumber + secondNumber;

console.log(sum);
```

- **Breakpoints**: Breakpoints let you pause the execution of your code at a specific line of your choice. After the pause, you can inspect variables, evaluate expressions, and examine the call stack.
- **Watchers**: Watch expressions lets you monitor the values of variables or expressions as the code runs even if they are out of the current scope.
- **Profiling**: Profiling helps you identify performance bottlenecks by letting you capture screenshots and record CPU usage, function calls, and execution time.
- `console.dir()`: This method is used to display an interactive list of the properties of a specified JavaScript object. It outputs a hierarchical listing that can be expanded to see all nested properties.

```
console.dir(document);
```

- `console.table()`: This method displays tabular data as a table in the console. It takes one mandatory argument, which must be an array or an object, and one optional argument to specify which properties (columns) to display.

Regular Expressions and Common Methods

- **Definition**: Regular Expressions, or Regex, are used to create a "pattern", which you can then use to check against a string, extract text, and more.

```
const regex = /freeCodeCamp/;
```

- `test()` **Method**: This method accepts a string, which is the string to test for matches against the regular expression. This method will return a boolean if the string matches the regex.

```
const regex = /freeCodeCamp/;
const test = regex.test("e");
console.log(test); // false
```

- `match()` **Method**: This method accepts a regular expression, although you can also pass a string which will be constructed into a regular expression. The `match` method returns the match array for the string.

```
const regex = /freeCodeCamp/;
const match = "freeCodeCamp".match(regex);
console.log(match); // ["freeCodeCamp"]
```

- `replace()` **Method**: This method accepts two arguments: the regular expression to match (or a string), and the string to replace the match with (or a function to run against each match).

```
const regex = /Jessica/;
const str = "Jessica is rly kewl";
const replaced = str.replace(regex, "freeCodeCamp");
console.log(replaced); // "freeCodeCamp is rly kewl"
```

- `replaceAll` **Method**: This method is used to replace all occurrences of a specified pattern with a new string. This method will throw an error if you give it a regular expression without the global modifier.

```
const text = "I hate JavaScript! I hate programming!";
const newText = text.replaceAll("hate", "love");
console.log(newText);  // "I love JavaScript! I love programming!"
```

- `matchAll` **Method**: This method is used to retrieve all matches of a given regular expression in a string, including capturing groups, and returns them as an iterator. An iterator is an object that allows you to go through (or "iterate over") a collection of items.

```
const str = "JavaScript, Python, JavaScript, Swift, JavaScript";
const regex = /JavaScript/g;

const iterator = str.matchAll(regex);

for (let match of iterator) {
  console.log(match[0]); // "JavaScript" for each match
}
```

Regular Expression Modifiers

- **Definition**: Modifiers, often referred to as "flags", modify the behavior of a regular expression.
- `i` **Flag**: This flag makes a regex ignore case.

```
const regex = /freeCodeCamp/i;
console.log(regex.test("freecodecamp")); // true
console.log(regex.test("FREECODECAMP")); // true
```

- `g` **Flag**: This flag, or global modifier, allows your regular expression to match a pattern more than once.

```
const regex = /freeCodeCamp/gi;
console.log(regex.test("freeCodeCamp")); // true
console.log(regex.test("freeCodeCamp is great")); // false
```

- **Anchor Definition**: The `^` anchor, at the beginning of the regular expression, says "match the start of the string". The `$` anchor, at the end of the regular expression, says "match the end of the string".

```
const start = /^freeCodeCamp/i;
const end = /freeCodeCamp$/i;
console.log(start.test("freecodecamp")); // true
console.log(end.test("freecodecamp")); // true
```

- `m` **Flag**: Anchors look for the beginning and end of the entire string. But you can make a regex handle multiple lines with the `m` flag, or the multi-line modifier.flag, or the multi-line modifier.

```
const start = /^freecodecamp/im;
const end = /freecodecamp$/im;
const str = `I love
freecodecamp
it's my favorite
`;
console.log(start.test(str)); // true
console.log(end.test(str)); // true
```

- `d` **Flag**: This flag expands the information you get in a match object.

```
const regex = /freecodecamp/di;
const string = "we love freecodecamp isn't freecodecamp great?";
console.log(string.match(regex));
```

- `u` **Flag**: This expands the functionality of a regular expression to allow it to match special unicode characters. The `u` flag gives you access to special classes like the `Extended_Pictographic` to match most emoji. There is also a `v` flag, which further expands the functionality of the unicode matching.
- `y` **Flag**: The sticky modifier behaves very similarly to the global modifier, but with a few exceptions. The biggest one is that a global regular expression will start from lastIndex and search the entire remainder of the string for another match, but a sticky regular expression will return null and reset the lastIndex to 0 if there is not immediately a match at the previous lastIndex.
- `s` **Flag**: The single-line modifier allows a wildcard character, represented by a `.` in regex, to match linebreaks - effectively treating the string as a single line of text.

Character Classes

- **Wildcard** `.`: Character classes are a special syntax you can use to match sets or subsets of characters. The first character class you should learn is the wild card class. The wild card is represented by a period, or dot, and matches ANY single character EXCEPT line breaks. To allow the wildcard class to match line breaks, remember that you would need to use the `s` flag.

```
const regex = /a./;
```

- `\d`: This will match all digits (`0-9`) in a string.

```
const regex = /\d/;
```

- `\w`: This is used to match any word character (`a-z0-9_`) in a string. A word character is defined as any letter, from a to z, or a number from 0 to 9, or the underscore character.

```
const regex = /\w/;
```

- `\s`: The white-space class `\s`, represented by a backslash followed by an `s`. This character class will match any white space, including new lines, spaces, tabs, and special unicode space characters.
- **Negating Special Character Classes**: To negate one of these character classes, instead of using a lowercase letter after the backslash, you can use the uppercase equivalent. The following example does not match a numerical character. Instead, it matches any single character that is NOT a numerical character.

```
const regex = /\D/;
```

- **Custom Character Classes**: You can create custom character classes by placing the character you wish match inside a set of square brackets.

```
const regex = /[abcdf]/;
```

Lookahead and Lookbehind Assertions

- **Definition**: Lookahead and lookbehind assertions allow you to match specific patterns based on the presence or lack of surrounding patterns.
- **Positive Lookahead Assertion**: This assertion will match a pattern when the pattern is followed by another pattern. To construct a positive lookahead, you need to start with the pattern you want to match. Then, use parentheses to wrap the pattern you want to use as your condition. After the opening parenthesis, use `?=` to define that pattern as a positive lookahead.

```
const regex = /free(?=code)/i;
```

- **Negative Lookahead Assertion**: This is a type of condition used in regular expressions to check that a certain pattern does not occur ahead in the string.

```
const regex = /free(?!code)/i;
```

- **Positive Lookbehind Assertion**: This assertion will match a pattern only if it is preceded by another specific pattern, without including the preceding pattern in the match.

```
const regex = /(?<=free)code/i;
```

- **Negative Lookbehind Assertion**: This assertion ensures that a pattern is not preceded by another specific pattern. It matches only if the specified pattern is not immediately preceded by the given sequence, without including the preceding sequence in the match.

```
const regex = /(?<!free)code/i;
```

Regex Quantifiers

- **Definition**: Quantifiers in regular expressions specify how many times a pattern (or part of a pattern) should appear. They help control the number of occurrences of characters or groups in a match. The following example is used to match the previous character exactly four times.

```
const regex = /^\d{4}$/;
```

- `*`: Matches 0 or more occurrences of the preceding element.
- `+`: Matches 1 or more occurrences of the preceding element.
- `?`: Matches 0 or 1 occurrence of the preceding element.
- `{n}`: Matches exactly n occurrences of the preceding element.
- `{n,}`: Matches n or more occurrences of the preceding element.
- `{n,m}`: Matches between n and m occurrences of the preceding element.

Capturing Groups and Backreferences

- **Capturing Groups**: A capturing group allows you to "capture" a portion of the matched string to use however you might need. Capturing groups are defined by parentheses containing the pattern to capture, with no leading characters like a lookahead.

```
const regex = /free(code)camp/i;
```

- **Backreferences**: A backreference in regular expressions refers to a way to reuse a part of the pattern that was matched earlier in the same expression. It allows you to refer to a captured group (a part of the pattern in parentheses) by its number. For example, `$1` refers to the first captured group.

```
const regex = /free(co+de)camp/i;
console.log("freecooooooooodecamp".replace(regex, "paid$1world"));
```

Validating Forms with JavaScript

- **Constraint Validation API**: Certain HTML elements, such as the `textarea` and `input` elements, expose a constraint validation API. This API allows you to assert that the user's provided value for that element passes any HTML-level validation you have written, such as minimum length or pattern matching.
- `checkValidity()` **method**: This method returns `true` if the element matches all HTML validation (based on its attributes), and `false` if it fails.

```
const input = document.querySelector("input");

input.addEventListener("input", (e) => {
  if (!e.target.checkValidity()) {
    e.target.setCustomValidity("You must use a .com email.")
  }
})
```

- `reportValidity()` **Method**: This method tells the browser that the `input` is invalid.

```
const input = document.querySelector("input");

input.addEventListener("input", (e) => {
  if (!e.target.checkValidity()) {
    e.target.reportValidity();
```

```
  }
})
```

- `validity` **Property**: This property is used to get or set the validity state of form controls (like `<input>`, `<select>`, etc.) and provides information about whether the user input meets the constraints defined for that element (e.g., `required` fields, pattern constraints, maximum length, etc.).

```
const input = document.querySelector("input");

input.addEventListener("input", (e) => {
  console.log(e.target.validity);
})
```

- `patternMismatch` **Property**: This will be `true` if the value doesn't match the specified regular expression pattern.

`preventDefault()` Method

- **Definition**: Every event that triggers in the DOM has some sort of default behavior. The click event on a checkbox toggles the state of that checkbox, by default. Pressing the space bar on a focused button activates the button. The `preventDefault()` method on these `Event` objects stops that behavior from happening.

```
button.addEventListener('click', (event) => {
  // Prevent the default button click behavior
  event.preventDefault();
  alert('Button click prevented!');
});
```

Submitting Forms

- **Definition**: There are three ways a form can be submitted. The first is when the user clicks a button in the form which has the `type` attribute set to `submit`. The second is when the user presses the `Enter` key on any editable `input` field in the form. The third is through a JavaScript call to the `requestSubmit()` or `submit()` methods of the `form` element.
- `action` **Attribute**: The `action` attribute should contain either a URL or a relative path for the current domain. This value determines where the form attempts to send data - if you do not set an `action` attribute, the form will send data to the current page's URL.

```
<form action="https://freecodecamp.org">
  <input
    type="number"
    id="input"
    placeholder="Enter a number"
    name="number"
  />
  <button type="submit">Submit</button>
</form>
```

- `method` **Attribute**: This attribute accepts a standard `HTTP` method, such as `GET` or `POST`, and uses that method when making the request to the action URL. When a method is not set, the form will default to a `GET` request. The data in the form will be URL encoded as `name=value` pairs and appended to the action URL as query parameters.

```
<form action="/data" method="POST">
  <input
    type="number"
    id="input"
    placeholder="Enter a number"
    name="number"
  />
  <button type="submit">Submit</button>
</form>
```

- `enctype` **Attribute**: The `form` element accepts an `enctype` attribute, which represents the encoding type to use for the data. This attribute only accepts three values: `application/x-www-form-urlencoded` (which is the default, sending the data as a URL-encoded form body), `text/plain` (which sends the data in plaintext form, in `name=value` pairs separated by new lines), or `multipart/form-data`, which is specifically for handling forms with a file upload.

The `date()` Object and Common Methods

- **Definition**: The `date()` object is used to create, manipulate, and format dates and times in JavaScript. In the following example, the `new` keyword is used to create a new instance of the `Date` object, and the `Date` object is then assigned to the variable `now`. If you were to log the value of `now` to the console, you would see the current date and time based on the system clock of the computer running the code.

```
const now = new Date();
```

- `Date.now()` **Method**: This method is used to get the current date and time. `Date.now()` returns the number of milliseconds since January 1, 1970, 00:00:00 UTC. This is known as the Unix epoch time. Unix epoch time is a common way to represent dates and times in computer systems because it is an integer that can be easily stored and manipulated. UTC stands for Universal Time Coordinated, which is the primary time standard by which the world regulates clocks and time.
- `getDate()` **Method**: This method is used to get a day of the month based on the current date. `getDate()` will return an integer value between 1 and 31, depending on the day of the month. If the date is invalid, it will return `NaN` (Not a Number).

```
const now = new Date();
const date = now.getDate();
console.log(date); // 15
```

- `getMonth()` **Method**: This method is used to get the month. The month is zero-based, so January is 0, February is 1, and so on. In this example, the output is 2, which corresponds to March. If the month is invalid, it will return `NaN`.

```
const now = new Date();
const month = now.getMonth();
console.log(month); // 2
```

- `getFullYear()` **Method**: This method is used to get the full year. If the year is invalid, it will return `NaN`.

```
const now = new Date();
const year = now.getFullYear();
console.log(year); // 2024
```

Different Ways to Format Dates

- `toISOString()` **Method**: This method is used to format the date in an extended `ISO` (ISO 8601) format. ISO 8601 is an international standard for representing dates and times. The format is `YYYY-MM-DDTHH:mm:ss.sssZ`.

```
const date = new Date();
console.log(date.toISOString());
```

- `toLocaleDateString()` **Method**: This method is used to format the date based on the user's locale.

```
const date = new Date();
console.log(date.toLocaleDateString());  // 11/23/2024
```

The `toLocaleDateString()` method accepts two optional parameters: locales and options.

The locales parameter is a string representing the locale to use. For example, you can pass in `"en-US"` for English (United States) or `"fr-FR"` for French (France). If you don't pass in a locales parameter, the default locale is used. The second optional parameter is the options parameter. This parameter is an object that allows you to specify the format of the date string.

```
const date = new Date();
const options = {
  weekday: "long",
  year: "numeric",
  month: "long",
  day: "numeric",
};
console.log(date.toLocaleDateString("en-GB", options)); // Saturday, November 23, 2024
```

`Audio` Constructor and Common Methods

- **Definition**: The `Audio` constructor, like other constructors, is a special function called with the `new` keyword. It returns an `HTMLAudioElement`, which you can then use to play audio for the user, or append to the DOM for the user to control themselves. When you call the constructor, you can optionally pass a URL as the (only) argument. This URL should point to the source of the audio file you want to play. Or, if you need to change the source dynamically, you can assign the URL to the `src` property of the returned audio element.
- `play()` **Method**: This method is used with the `audio` or `video` elements to begin playback for the media.

```
const audio = document.getElementById('audio');
```

```
// Starts playing the audio
audio.play();
```

- `pause()` **Method**: This method is used with the `audio` or `video` elements to pause playback for the media.

```
function pauseAudio() {
  const audio = document.getElementById('myAudio');
  audio.pause();  // Pauses the audio playback
}
```

- `addTextTrack()` **Method**: This method allows you to specify a text track to associate with the media element - which is especially helpful for adding subtitles to a video.
- `fastSeek()` **Method**: This method allows you to move the playback position to a specific time within the media.

Different Audio and Video Formats

- **MIME type**: A MIME type, standing for Multipurpose Internet Mail Extensions, is a standardized way to programmatically indicate a file type. The MIME type can tell an application, such as your browser, how to handle a specific file. In the case of audio and video, the MIME type indicates it is a multimedia format that can be embedded in the web page.
- `source` **Element**: This is used to specify a file type and source - and can include multiple different types by using multiple source elements. When you do this, the browser will determine the best format to use for the user's current environment.
- **MP3**: This is a type of digital file format used to store music, audio, or sound. It's a compressed version of a sound recording that makes the file size smaller, so it's easier to store and share. An MP3 file has the MIME type audio/mp3
- **MP4**: An MP4 is a type of digital file format used to store video and audio. It serves as a container that holds both the video (images) and the sound (music or speech) in one file. An MP4, can have the MIME type audio/mp4 OR video/mp4, depending on whether it's a video file or audio-only.

codecs

- **Definition**: A codec, short for "encoder/decoder", is an algorithm or software that can convert audio and video between analogue and digital formats. Codecs can be specified as part of the MIME type. The basic syntax to define a codec is to add a semi-colon after the media type, then `codecs=` and the codec.

HTMLMediaElement API

- **Definition**: The `HTMLMediaElement` API is used to control the behavior of audio and video elements on your page. It extends the base `HTMLElement` interface, so you have access to the base properties as well as these helpful methods. Examples of these methods include `play()`, `fastSeek()`, and `pause()`.

Media Capture and Streams API

- **Definition**: The Media Capture and Streams API, or the MediaStream API, is used to capture audio and video from your device. In order to use the API, you need to create the `MediaStream` object. You could do this with the constructor, but it would not be tied to the user's hardware. Instead, the `mediaDevices` property of the `global` navigator object has a `getUserMedia()` method for you to use.

```
window.navigator.mediaDevices.getUserMedia({
  audio: true,
  video: {
    width: {
      min: 1280,
      ideal: 1920,
```

```
      max: 3840
    },
    height: {
      min: 720,
      ideal: 1080,
      max: 2160
    }
  }
});
```

Screen Capture API

- **Definition**: The Screen Capture API is used to record a user's screen. This API is exposed by calling the `getDisplayMedia()` method of the `mediaDevices` object and consuming the returned media stream.

MediaStream Recording API

- **Definition**: The MediaStream Recording API works in tandem with the MediaStreams APIs, allowing you to record a MediaStream (or even an `HTMLMediaElement` directly).

Media Source Extensions API

- **topic**: The Media Source Extensions API is what allows you to directly pass a user's webcam feed to a video element with the `srcObject` property.

Web Audio API

- **Definition**: The Web Audio API which powers everything audible on the web. This API includes important objects like an `AudioBuffer` (representing a Buffer specifically containing audio data) or the `AudioContext`.

Sets in JavaScript

- A `Set` is a built-in option for managing data collection.
- Sets ensure that each value in it appears only once, making it useful for eliminating duplicates from an array or handling collections of distinct values.
- You can create a `Set` using the `Set()` constructor:

```
const set = new Set([1, 2, 3, 4, 5]);
console.log(set); // Set { 1, 2, 3, 4, 5 }
```

- Sets can be manipulated using these methods:
  - `add()`: Adds a new element to the `Set`.
  - `delete()`: Removes an element from the `Set`.
  - `has()`: Checks if an element exists in the `Set`.
  - `clear()`: Removes all elements from the `Set`.

Weaksets in JavaScript

- `WeakSet` is a collection of objects that allows you to store weakly held objects.

Sets vs WeakSets

- Unlike Sets, a `WeakSet` does not support primitives like numbers or strings.
- A `WeakSet` only stores objects, and the references to those objects are "weak," meaning that if the object is not being used anywhere else in your code, it is removed automatically to free up memory.

Maps in JavaScript

- A `Map` is a built-in object that holds key-value pairs just like an object.
- Maps differ from the standard JavaScript objects with their ability to allow keys of any type, including objects, and functions.
- A `Map` provides better performance over the standard object when it comes to frequent addition and removals of key-value pairs.
- You can create a `Map` using the `Map()` constructor:

```
const map = new Map([
  ['flower', 'rose'],
  ['fruit', 'apple'],
  ['vegetable', 'carrot']
]);
console.log(map); // Map(3) { 'flower' => 'rose', 'fruit' => 'apple', 'vegetable' => 'carrot' }
```

- Maps can be manipulated using these methods:
    - `set()`: Adds a new key-value pair to the `Map`.
    - `get()`: Retrieves the value of a key from the `Map`.
    - `delete()`: Removes a key-value pair from the `Map`.
    - `has()`: Checks if a key exists in the `Map`.
    - `clear()`: Removes all key-value pairs from the `Map`.

WeakMaps in JavaScript

- A `WeakMap` is a collection of key-value pairs just like `Map`, but with weak references to the keys. The keys must be an object and the values can be anything you like.

Maps vs WeakMaps

- WeakMaps are similar to WeakSets in that they only store objects and the references to those objects are "weak."

Persistent Storage

- **Definition**: Persistent storage refers to a way of saving data in a way that it stays available even after the power is turned off or the device is restarted.

Create, Read, Update, Delete (CRUD)

- **Create**: This refers to the process of creating new data. For example, in a web app, this could be when a user adds a new post to a blog.
- **Read**: This is the operation where data is retrieved from a database. For instance, when you visit a blog post or view your profile on a website, you're performing a read operation to fetch and display data stored in the database.
- **Update**: This involves modifying existing data in the database. An example would be editing a blog post or updating your profile information.
- **Delete**: This is the operation that removes data from a database. For instance, when you delete a blog post or account, you're performing a delete operation.

HTTP Methods

- **Definition**: HTTP stands for Hypertext Transfer Protocol and it is the foundation for data communication on the web. There are HTTP methods which define the actions that can be performed on resources over the web. The common methods are GET, POST, PUT, PATCH, DELETE.
- `GET` **Method**: This is used to fetch data from a server.
- `POST` **Method**: This is used to submit data to a server which creates a new resource.
- `PUT` **Method**: This is used to update a resource by replacing it entirely.
- `PATCH` **Method**: This is used to partially update a resource.
- `DELETE` **Method**: This is used to remove records from a database.

`localStorage` and `sessionStorage` Properties

- **Web Storage API**: This API provides a mechanism for browsers to store key-value pairs right within the browser, allowing developers to store information that can be used across different page reloads and sessions. The two main components for the Web Storage API are the `localStorage` and `sessionStorage` properties.
- `localStorage` **Property**: `localStorage` is the part of the Web Storage API that allows data to persist even after the browser window is closed or the page is refreshed. This data remains available until it is explicitly removed by the application or the user.
- `localStorage.setItem()` **Method**: This method is used to store a key-value pair in `localStorage`.

```
localStorage.setItem('username', 'Jessica');
```

- `localStorage.getItem()` **Method**: This method is used to retrieve the value of a given key from `localStorage`.

```
localStorage.setItem('username', 'codingRules');

let username = localStorage.getItem('username');
console.log(username); // codingRules
```

- `localStorage.removeItem()` **Method**: This method is used to remove a specific item from `localStorage` using its key.

```
localStorage.removeItem('username');
```

- `localStorage.clear()` **Method**: This method is used to clear all of the stored data in `localStorage`.

```
localStorage.clear();
```

- `sessionStorage` **Property**: Stores data that lasts only for the current session and is cleared when the browser tab or window is closed.
- `sessionStorage.setItem()` **Method**: This method is used to store a key-value pair in `sessionStorage`.

```
sessionStorage.setItem('cart', '3 items');
```

- `sessionStorage.getItem()` **Method**: This method is used to retrieve the value of a given key from `sessionStorage`.

```
sessionStorage.setItem('cart', '3 items');
```

```
let cart = sessionStorage.getItem('cart');
console.log(cart); // '3 items'
```

- `sessionStorage.removeItem()` **Method**: This method is used to remove a specific item from `sessionStorage` using its key.

```
sessionStorage.removeItem('cart');
```

- `sessionStorage.clear()` **Method**: This method is used to clear all data stored in `sessionStorage`.

```
sessionStorage.clear();
```

Working with Cookies

- **Definition**: Cookies, also known as web cookies or browser cookies, are small pieces of data that a server sends to a user's web browser. These cookies are stored on the user's device and sent back to the server with subsequent requests. Cookies are essential in helping web applications maintain state and remember user information, which is especially important since HTTP is a stateless protocol.
- **Session Cookies**: These cookies only last for the duration of the user's session on the website. Once the user closes the browser or tab, the session cookie is deleted. These cookies are typically used for tasks like keeping a user logged in during their visit.
- **Secure Cookies**: These cookies are only sent over HTTPS, ensuring that they cannot be intercepted by an attacker in transit.
- **HttpOnly Cookies**: These cookies cannot be accessed or modified by JavaScript running in the browser, making them more secure against cross-site scripting (XSS) attacks.
- **Set-Cookie Header**: When you visit a website, the server can send a Set-Cookie header in the HTTP response. This header tells your browser to save a cookie with specific information. For example, it might store a unique ID that helps the site recognize you the next time you visit.

You can manually set a cookie in JavaScript using `document.cookie`:

```
document.cookie = "organization=freeCodeCamp; expires=Fri, 31 Dec 2021 23:59:59 GMT; path=/";
```

To delete a cookie, you can set its expiration date to a time in the past.

```
document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 GMT; path=/";
```

Cache API

- **Definition**: Caching is the process of storing copies of files in a temporary storage location, so that they can be accessed more quickly. The Cache API is used to store network requests and responses, making web applications work more efficiently and even function offline. It is part of the broader Service Worker API and is crucial for creating Progressive Web Apps (PWAs) that can work under unreliable or slow network conditions.

The Cache API is a storage mechanism that stores Request and Response objects. When a request is made to a server, the application can store the response and later retrieve it from the cache instead of making a new network request. This reduces load times, saves bandwidth, and improves the overall user experience.

- **Cache Storage**: This is used to store key-value pairs of HTTP requests and their corresponding responses. This enables efficient retrieval of previously requested resources, reducing the need to fetch them from the network on subsequent visits and improving performance.
- **Cache-Control**: Developers can define how long a cached resource should be kept, and if it should be revalidated or served directly from cache.
- **Offline Support**: By using the Cache API, you can create offline-first web applications. For example, a PWA can serve cached assets when the user is disconnected from the network.

Negative Patterns and Client Side Storage

- **Excessive Tracking**: This refers to the practice of collecting and storing an overabundance of user data in client-side storage (such as cookies, local storage, or session storage) without clear, informed consent or a legitimate need. This often involves tracking user behavior, preferences, and interactions across multiple sites or sessions, which can infringe on user privacy.
- **Browser Fingerprinting**: A technique used to track and identify individual users based on unique characteristics of their device and browser, rather than relying on cookies or other traditional tracking methods. Unlike cookies, which are stored locally on a user's device, fingerprinting involves collecting a range of information that can be used to create a distinctive "fingerprint" of a user's browser session.
- **Setting Passwords in LocalStorage**: This might seem like a more obvious negative pattern, but setting any sensitive data like passwords in local storage posses a security risk. Local Storage is not encrypted and can be accessed easily. So you should never store any type of sensitive data in there.

IndexedDB

- **Definition**: IndexedDB is used for storing structured data in the browser. This is built into modern web browsers, allowing web apps to store and fetch JavaScript objects efficiently.

Cache/Service Workers

- **Definition**: A Service Worker is a script that runs in the background which is separate from your web page. It can intercept network requests, access the cache, and make the web app work offline. This is a key component of Progressive Web Apps.

Basics of Working with Classes

- **Definition**: Classes in JavaScript are used to define blueprints for creating objects, and encapsulating data. Classes include a constructor which is a special method that gets called automatically when a new object is created from the class. It is used to initialize the properties of the object. The `this` keyword is used here to refer to the current instance of the class. Below the constructor, you can have what are called methods. Methods are functions defined inside a class that perform actions or operations on the class's data or state. They are used to define behaviors that instances of the class can perform.

```
class Dog {
  constructor(name) {
    this.name = name;
  }

  bark() {
    console.log(`${this.name} says woof!`);
  }
}
```

To create a new instance of the class, you will use the `new` keyword followed by the class name:

```
const dog = new Dog("Gino");
```

You can also create classes as class expressions. This is where the class is anonymous and assigned to a variable.

```
const Dog = class {
  constructor(name) {
```

```
    this.name = name;
  }

  bark() {
    console.log(`${this.name} says woof!`);
  }
};
```

Class Inheritance

- **Definition**: In programming, inheritance allows you to define classes that inherit properties and methods from parent classes. This promotes code reuse and establishes a hierarchical relationship between classes. A parent class is a class that acts like a blueprint for other classes. It defines properties and methods that are inherited by other classes. A child class is a class that inherits the properties and methods of another class. Child classes can also extend the functionality of their parent classes by adding new properties and methods. In JavaScript, we use the `extends` keyword to implement inheritance. This keyword indicates that a class is the child class of another class.

```
class Vehicle {
  constructor(brand, year) {
    this.brand = brand;
    this.year = year;
  }
}

class Car extends Vehicle {
  honk() {
    console.log("Honk! Honk!");
  }
}
```

The `super` keyword is used to access the parent class's methods, constructors, and fields.

```
class Vehicle {
  constructor(brand, year) {
    this.brand = brand;
    this.year = year;
  }
}

class Car extends Vehicle {
  constructor(brand, year, numDoors) {
    super(brand, year);
    this.numDoors = numDoors;
```

```
  }
}
```

Working with Static Methods and Static Properties

- **Static methods**: These methods are often used for utility functions that don't need access to the specific state of an object. They are defined within classes to encapsulate related functionality. Static methods are also helpful for implementing "factory" methods. A factory method is a method that you define in addition to the constructor to create objects based on specific criteria.

```
class Movie {
  constructor(title, rating) {
    this.title = title;
    this.rating = rating;
  }

  static compareMovies(movieA, movieB) {
    if (movieA.rating > movieB.rating) {
      console.log(`${movieA.title} has a higher rating.`);
    } else if (movieA.rating < movieB.rating) {
      console.log(`${movieB.title} has a higher rating.`);
    } else {
      console.log("These movies have the same rating.");
    }
  }
}

let movieA = new Movie("Movie A", 80);
let movieB = new Movie("Movie B", 45);

Movie.compareMovies(movieA, movieB);
```

- **Static Properties**: These properties are used to define values or attributes that are associated with a class itself, rather than with instances of the class. Static properties are shared across all instances of the class and can be accessed without creating an instance of the class.

```
class Car {
  // Static property
  static numberOfWheels = 4;

  constructor(make, model) {
    this.make = make;
    this.model = model;
  }
```

```
  // Instance method
  getCarInfo() {
    return `${this.make} ${this.model}`;
  }

  // Static method
  static getNumberOfWheels() {
    return Car.numberOfWheels;
  }
}

// Accessing static property directly from the class
console.log(Car.numberOfWheels);
```

- Recursion is programming concept that allows you to call a function repeatedly until a base-case is reached.

Here is an example of a recursive function that calculates the factorial of a number:

```
function findFactorial(n) {
  if (n === 0) {
    return 1;
  }
  return n * findFactorial(n - 1);
}
```

In the above example, the `findFactorial` function is called recursively until `n` reaches `0`. When `n` is `0`, the base case is reached and the function returns `1`. The function then returns the product of `n` and the result of the recursive call to `findFactorial(n - 1)`.

- Recursion allows you to handle something with an unknown depth, such as deeply nested objects/arrays, or a file tree.
- A call stack is used to keep track of the function calls in a recursive function. Each time a function is called, it is added to the call stack. When the base case is reached, the function calls are removed from the stack.
- You should carefully define the base case as calling it indefinitely can cause your code to crash. This is because the recursion keeps piling more and more function calls till the system runs out of memory.
- Recursions find their uses in solving mathematical problems like factorial and Fibonacci, traversing trees and graphs, generating permutations and combinations and much more.

Pure vs Impure Functions

- A pure function is one that always produces the same output for the same input and doesn't have any side effects. Its output depends only on its input, and it doesn't modify any external state.
- Impure functions have side effects, which are changes to the state of the program that are observable outside the function.

Functional programming

- Functional Programming is an approach to software development that emphasizes the use of functions to solve problems, focusing on what needs to be done rather than how to do it.

- Functional programming encourages the use of techniques that help avoid side effects, such as using immutable data structures and higher-order functions.
- When used correctly, functional programming principles lead to cleaner and more maintainable code

Currying

- Currying is a functional programming technique that transforms a function with multiple arguments into a sequence of functions, each taking a single argument.

Here is an example of a regular function vs a curried function:

```
// Regular function

function average(a, b, c) {
  return (a + b + c) / 3;
}

// Curried function

function curriedAverage(a) {
  return function(b) {
    return function(c) {
      return (a + b + c) / 3;
    };
  };
}

// Usage of curried function

const avg = curriedAverage(2)(3)(4);
```

- Currying can be particularly powerful when working with functions that take many arguments.
- Currying makes your code more flexible and easier to reuse.
- You can use arrow functions to create curried functions more concisely:

```
const curriedAverage = a => b => c => (a + b + c) / 3;
```

- While currying can lead to more flexible and reusable code, it can also make code harder to read if overused.
- **Synchronous JavaScript** is executed sequentially and waits for the previous operation to finish before moving on to the next one.
- **Asynchronous JavaScript** allows multiple operations to be executed in the background without blocking the main thread.
- **Thread** is a sequence of instructions that can be executed independently of the main program flow.
- **Callback functions** are functions that are passed as arguments to other functions and are executed after the completion of the operation or as a result of an event.

The JavaScript engine and JavaScript runtime

- The **JavaScript engine** is a program that executes JavaScript code in a web browser. It works like a converter that takes your code, turns it into instructions that the computer can understand and work accordingly.
- V8 is an example of a JavaScript engine developed by Google.
- The **JavaScript runtime** is the environment in which JavaScript code is executed. It includes the JavaScript engine which processes and executes the code, and additional features like a web browser or Node.js.

The Fetch API

- The Fetch API allows web apps to make network requests, typically to retrieve or send data to the server. It provides a `fetch()` method that you can use to make these requests.
- You can retrieve text, images, audio, JSON, and other types of data using the Fetch API.

HTTP methods for Fetch API

The Fetch API supports various HTTP methods to interact with the server. The most common methods are:

- **GET**: Used to retrieve data from the server. By default, the Fetch API uses the `GET` method to retrieve data.

```
fetch('https://api.example.com/data')
```

To use the fetched data, it must be converted to JSON format using the `.json()` method:

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
```

In this code, the response coming from the Fetch API is a promise and the `.then` handler is converting the response to a JSON format.

- **POST**: Used to send data to the server. The `POST` method is used to create new resources on the server.

```
fetch('https://api.example.com/users', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    name: 'John Doe',
    email: 'john@example.com'
  })
})
```

In this example, we're sending a `POST` request to create a new user. We have specified the method as `POST`, set the appropriate headers, and included a body with the data we want to send. The body needs to be a string, so we use `JSON.stringify()` to convert our object to a JSON string.

- **PUT**: Used to update data on the server. The `PUT` method is used to update existing resources on the server.

```
fetch('https://api.example.com/users/45', {
  method: 'PUT',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    name: 'John Smith',
    email: 'john@example.com'
  })
})
```

In this example, we are updating the ID `45` that is specified at the end of the URL. We have used the `PUT` method on the code and also specified the data as the body which will be used to update the identified data.

- **DELETE**: Used to delete data on the server. The `DELETE` method is used to delete resources on the server.

```
fetch('https://api.example.com/users/45', {
  method: 'DELETE'
})
```

In this example, we're sending a `DELETE` request to remove a user with the ID `45`.

Promise and promise chaining

- **Promises** are objects that represent the eventual completion or failure of an asynchronous operation and its resulting value. The value of the promise is known only when the `async` operation is completed.
- Here is an example to create a simple promise:

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Data received successfully');
  }, 2000);
});
```

- The `.then()` method is used in a Promise to specify what should happen when the Promise is fulfilled, while `.catch()` is used to handle any errors that occur.
- Here is an example of using `.then()` and `.catch()` with a Promise:

```
promise
  .then(data => {
    console.log(data);
  })
  .catch(error => {
```

```
    console.error(error);
  });
```

In the above example, the `.then()` method is used to log the data received from the Promise, while the `.catch()` method is used to log any errors that occur.

- **Promise chaining**: One of the powerful features of Promises is that we can chain multiple asynchronous operations together. Each `.then()` can return a new Promise, allowing you to perform a sequence of asynchronous operations one after the other.
- Here is an example of Promise chaining:

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => {
    console.log(data);
    return fetch('https://api.example.com/other-data');
  })
  .then(response => response.json())
  .then(otherData => {
    console.log(otherData);
  })
  .catch(error => {
    console.error(error);
  });
```

In the above example, we first fetch data from one URL, then fetch data from another URL based on the first response, and finally log the second data received.

The `catch` method would handle any errors that occur during the process. This means you don't need to add error handling to each individual step, which can greatly simplify your code.

Using `async/await` to handle promises

Async/await makes writing & reading asynchronous code easier which is built on top of Promises.

- **async**: The `async` keyword is used to define an asynchronous function. An `async` function returns a Promise, which resolves with the value returned by the `async` function.
- **await**: The `await` keyword is used inside an `async` function to pause the execution of the function until the Promise is resolved. It can only be used inside an `async` function.
- Here is an example of using `async/await`:

```
async function delayedGreeting(name) {
  console.log("A Messenger entered the chat...");
  await new Promise(resolve => setTimeout(resolve, 2000));
  console.log(`Hello, ${name}!`);
}
```

```
delayedGreeting("Alice");
console.log("First Printed Message!");
```

In the above example, the `delayedGreeting` function is an `async` function that pauses for 2 seconds before printing the greeting message. The `await` keyword is used to pause the function execution until the `Promise` is resolved.

- One of the biggest advantages of `async/await` is error handling via `try/catch` blocks. Here's an example:

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}

fetchData();
```

In the above example, the `try` block contains the code that might throw an error, and the `catch` block handles the error if it occurs. This makes error handling more straightforward and readable.

The `async` attribute

- The `async` attribute tells the browser to download the script file asynchronously while continuing to parse the HTML document.
- Once the script is downloaded, the HTML parsing is paused, the script is executed, and then HTML parsing resumes.
- You should use `async` for independent scripts where the order of execution doesn't matter

The `defer` attribute

- The `defer` attribute also downloads the script asynchronously, but it defers the execution of the script until after the HTML document has been fully parsed.
- The `defer` scripts maintain the order of execution as they appear in the HTML document.
- It's important to note that both `async` and `defer` attributes are ignored for inline scripts and only work for external script files.
- When both `async` and `defer` attributes are present, the `async` attribute takes precedence.

Geolocation API

- The Geolocation API provides a way for websites to request the user's location.
- The example below demonstrates the API's `getCurrentPosition()` method which is used to get the user's current location.

```
navigator.geolocation.getCurrentPosition(
  (position) => {
    console.log("Latitude: " + position.coords.latitude);
```

```
    console.log("Longitude: " + position.coords.longitude);
  },
  (error) => {
    console.log("Error: " + error.message);
  }
);
```

In this code, we're calling `getCurrentPosition` and passing it a function which will be called when the position is successfully obtained.

The `position` object contains a variety of information, but here we have selected `latitude` and `longitude` only.

If there is an issue with getting the `position`, then the error will be logged to the console. It is important to respect the user's privacy and only request their location when necessary.