js Notes

- JavaScript (JS) is a dynamic programming/scripting language that allows you to create logic depending on your logical ability. It also stores values of variables irrespective of their type.
- JS is a synchronous programming language, meaning that execution is done in order. It is also a single-threaded programming language, which means that the JS engine can only execute one statement at a time, and tasks are done one by one.
- Unlike compiled programming languages, JS is not compiled all at once and converted to low-level language (binary or machine code) that is read by the microprocessor. Instead, it is an interpreted programming language, which means that the code is executed line by line. If an error occurs at a line, the code stops processing further.
- Every browser has two engines: a layout engine (used to process HTML and CSS) and a JS engine (used to process the dynamic part of JS).
- JavaScript is a weakly typed programming language, meaning you can change the data type of a variable after defining it. For example:

```
var a = 10;
a = "abc";
```

Note -The console is a REPL, i.e., a read-evaluate-print loop. negative indexing is possible in case of js it starts from last i.e -3,-2,-1 and Code flow is from top to bottom and left to right

Data types :

Primitive data types (lower level/predefined dataType) in JavaScript include string, Boolean, number, null, undefined, BigInt, and symbol.

String :

String methods :

- **Substr**: This method takes two arguments. The first argument is the starting point in the string from where it begins, and it continues to the end of the string. The second argument is the length of the substring you want to extract.
- **Substring**: This method replaces `substr`. It takes two arguments: the starting point and the ending point. However, the character at the ending point is not included in the result. If the starting point is greater than the ending point, they are swapped. If negative values are provided, they are treated as zero.
- **IndexOf**: This function searches for a specific character or substring within a string. It takes one argument, which is the character or substring to search for. It returns the index (position) of the first occurrence of the search term. You can optionally provide a second argument as the ending index to start searching from a specific point. This method can be used with arrays as well. If the item is not found, it returns -1. It's a non-destructive method.
- **Replace**: The `replace` method takes two arguments. The first argument specifies what you want to replace, and the second argument is what you want to replace it with. It only replaces the first occurrence of the specified value.
- **ReplaceAll**: Similar to `replace`, but it replaces all occurrences of the specified value in the string.
- **Repeat**: This method repeats the string a specified number of times, based on the argument you provide.
- **ToUpperCase**: It converts all characters in the string to uppercase.
- **ToLowerCase**: It converts all characters in the string to lowercase.
- **Trim**: `Trim` removes any extra spaces (whitespace) from both the beginning and the end of the string.
- **Split**: The `split` method takes an argument that specifies the character or substring around which the string should be split. For example, `str.split('-')` will return an array containing the parts of the string that were separated by hyphens. This method is non-destructive.

Typeof- is a method that gives/tells the type of variable
Note - 0/0 is not a number but typeof this gives number as typeOf(NaN) is number(Infinity, NaN, -ve, +ve all these are Number type)

Math Object :

The Math object is a valuable tool for performing mathematical operations in JavaScript. It provides a variety of methods to assist with calculations, such as:

- `Math.PI`: This constant represents the mathematical constant Pi, approximately 3.14159.
- `Math.sqrt()`: Used to find the square root of a number.
- `Math.round()`: Rounds a number to the nearest integer.
- `Math.min()` and `Math.max()`: These functions help find the smallest and largest values in a set of numbers, respectively.
- `Math.pow()`: Computes the power of a number.
- `Math.ceil()`: Rounds a number up to the nearest integer. If the number is already an integer, it remains unchanged.
- `Math.floor()`: Rounds a number down to the nearest integer, effectively removing any decimal part.
- `Math.random()`: Generates a pseudo-random decimal number between 0 (inclusive) and 1 (exclusive). To extend the range, you can multiply this value by the desired range.

For example, if you want a random number between 22 and 27, you can achieve this as follows:

1. Start with the base value, which is 22, as your lower limit.
2. To create a random value between 0 and 1 (excluding 1), you can use `Math.random()`.
3. To expand this range to 0-5, you can multiply the random value by 5 (`Math.random() * 5`).
4. Finally, by adding this to your lower limit (22 + `Math.random() * 5`), you'll get a random number between 22 and 27.

This allows you to generate random numbers within specific ranges for various purposes in your JavaScript code.

Conditionals :

Conditionals in JavaScript are used to make decisions in your code. They allow you to execute different blocks of code based on whether a specified condition evaluates to true or false. JavaScript provides several conditional statements and operators to control the flow of your program. Here are some common ones:

1. **if Statement**: The "if" statement is used when a condition stands alone or is self-dependent.

```
javascriptCopy code
if (condition) {
    // Code to be executed if the condition is true
}
```

2. **if-else Statement**:

```
javascriptCopy code
if (condition) {
    // Code to be executed if the condition is true
} else {
    // Code to be executed if the condition is false
}
```

3. **else-if Statement**:

```
javascriptCopy code
if (condition1) {
    // Code to be executed if condition1 is true
} else if (condition2) {
    // Code to be executed if condition2 is true
} else {
    // Code to be executed if neither condition1 nor condition2 is true
}
```

4. **switch Statement**:

```
javascriptCopy code
switch (expression) {
    case value1:
        // Code to be executed if expression matches value1
        break;
    case value2:
        // Code to be executed if expression matches value2
        break;
    default:
        // Code to be executed if expression doesn't match any case
}
```

5. **Ternary Operator (Conditional Operator)**:

```
javascriptCopy code
var result = (condition) ? trueValue : falseValue;
```

6. **Logical Operators** (`&&`, `||`, `!`) for combining conditions.

```
var age = 25;
if (age < 18) {
    console.log("You are a minor.");
} else if (age >= 18 && age < 65) {
    console.log("You are an adult.");
} else {
    console.log("You are a senior citizen.");
}
```

These conditional statements and operators help you control the flow of your JavaScript code and execute different actions based on various conditions.

Operators :

Logical Operators:

- **&& (Logical AND):** This operator returns true if both operands are true; otherwise, it returns false.
- **|| (Logical OR):** This operator returns true if at least one of the operands is true; otherwise, it returns false.
- **! (Logical NOT):** This operator negates the value of its operand. If the operand is true, it returns false, and if the operand is false, it returns true.

Relational Operators:

- **!= (Not Equal To):** This operator checks if two values are not equal.
- **!== (Strict Not Equal To):** This operator checks if two values are not equal without performing type conversion.
- **=== (Strict Equal To):** This operator checks if two values are strictly equal without performing type conversion.
- **== (Equal To):** This operator checks if two values are equal, performing type conversion if necessary.
- **<= (Less Than or Equal To):** This operator checks if the left operand is less than or equal to the right operand.
- **>= (Greater Than or Equal To):** This operator checks if the left operand is greater than or equal to the right operand.
- **< (Less Than):** This operator checks if the left operand is less than the right operand.
- **> (Greater Than):** This operator checks if the left operand is greater than the right operand.

These operators are essential for performing logical and relational comparisons in JavaScript.

Notes:

- *Prompt Usage:*
  - *Prompt is utilized to obtain user input via an alert.*
- *Type Conversion:*

  - *To convert a value to an integer, you can use `parseInt`.*
  - *To convert it to a floating-point number, you can use `parseFloat`.*
- *JavaScript Objects:*
  - *It's important to note that in JavaScript, everything is treated as an object.*

Array :

An array is an ordered data structure, meaning the sequence in which elements are added is preserved. Array is Hetrogeneous, meaning it can store various types of data, even if they are dissimilar.
You can use the `.length` property to determine how many elements are in an array.
Array methods come in two types: destructive (they permanently modify the original array) and non-destructive.

- `.push`: Adds an element to the end of the array and returns the new length. This is a destructive method.
- `.pop`: Removes an element from the end of the array and returns that element. This is a destructive method.
- `.unshift`: Similar to `.push`, but it operates on the first element of the array. This is a destructive method.
- `.shift`: Similar to `.pop`, but it operates on the first element of the array. This is a destructive method.
- `.slice`: Returns a portion of the array starting from a specified index (inclusive) to an ending index (exclusive) and takes these 2 as the arguments. It's a non-destructive method.
- `.splice`: Removes elements from the array starting at a given index and returns them. It's a destructive method. the 2nd argument that it takes is length.Additional arguments can be provided to insert elements at the removal index.
- `.join`: Joins array elements into a string with a specified separator and returns the string. It's a non-destructive method.

- `.includes`: Checks if an array contains a specified element and returns a Boolean. It's a non-destructive method and can be used with strings.
- `.concat`: Combines two arrays. It's a non-destructive method and works with strings too.
- `.reverse`: Reverses the elements of an array. It's a destructive method.
- `.forEach`: Iterates over each element of an array and applies a callback function. It doesn't return anything and is non-destructive. Eg.- arr.forEach(function (item){clg(item)}) in this function accepts arguments 1st as the iterable items and 2nd is index of the item
- `.map`: Creates(returns) a new array by applying a callback function to each element. It's non-destructive method.Eg.-let c= arr.map(function(item){return item*2}), its function can take 2 arguments just like ForEach item and index and we can return both using return [item, index].
- `.filter`: Constructs a new array with elements that pass a test provided by a callback function. It doesn't return index, only the item.Eg.- let z= arr.filter (function (item){if(item >4){return true} return false;})
- `.sort`: Sorts the array lexicographically i.e-Sort on basis of its index of characters i.e. start from 1 then 2 and so on 11,2,45,66,7,8,90,93. Sort accepts a callback function. You can pass a comparator order for custom sorting (ascending or descending and non-lexicographical sort) i.e.- if a has to be greater than b for asec order then a-b is returned as a-b should be greater than 0 ||ly for desc b-a, syntax to achieve this is by passing comparator function(which will work as callback in case of sort) as an argument - arr.sort(function (a,b){return a-b});

Arrays are reference types, so if you set one array equal to another, both point to the same memory location. Changing one will affect the other.

Object :

An unordered data structure means that the order of values stored in it may not necessarily be the same as the order they were added. It serves as a collection of properties i.e. pairs, consisting of keys and their corresponding values. To access the properties of an object within this structure, we utilize the "dot" operator or square brackets ([]). Notably, keys are stored as strings in memory. This data structure is also considered a reference type.

Note-

- *Methods are functions that are defined within an object.*
- *The function always returns a value. If nothing is explicitly returned, it defaults to returning 'undefined'.*
- *The 'this' keyword in a method is used to access properties of the object. 'this' points to the object that has the method.*

Loop Types:

1. **for Loop:**
   - Commonly used for iterating over arrays.
   - Syntax: `for (let i = 0; i < array.length; i++) { }`
2. **do-while Loop:**
   - Executes code block at least once, then checks the condition.
   - Syntax: `do { } while (condition);`
3. **while Loop:**
   - Repeats code block while a condition is true.
   - Syntax: `while (condition) { }`
4. **for-in Loop:**
   - Used to loop through object properties (keys).
   - Syntax: `for (let key in object) { }`
   - Access values with `object[key]`.
5. **for-of Loop:**
   - Iterates over iterable items (e.g., arrays, maps).

    - Syntax: `for (let item of iterable) { }`
    - Access values directly using `item`.

These are different types of loops in JavaScript and how to use them.

Functions are the heart of Js. types- Function declaration, function calling, parametrized function, functional expression, first-class function, anonymous function, higher-order function, callback functions

Function declaration:

- Functions are defined using the `function` keyword, followed by a name, e.g., `function myFunction(){}`.
- A function must be called to execute its code; without a function call, the function remains inactive.
- **Function Parameters**: In the function declaration, values enclosed in `()` are called parameters, e.g., `function greet(name){}` where `name` is a parameter.
- **Function Arguments**: When calling a function, values passed inside the `()` are called arguments, e.g., `greet("John")` where "John" is an argument.
- A function can be parametrized by receiving values through its parameters when called.
- Functions can have default parameter values. If no argument is provided, the parameter takes its default value.
- If an argument is given, it overrides the default value assigned in the parameter definition.

Note- Function calling is to call a defined function, And without a function call, the function will not run

Functional expression:

A functional expression is a concept in programming where you store a function within a variable. It's often referred to as a "first-class function" or a "first-class citizen" in the world of programming.

```
const greet = function(name) {
    return `Hello, ${name}!`;
};
```

- Here, the `greet` variable holds a function that can be used just like any other variable.
- Functional expressions are powerful because they allow you to work with functions in a flexible and dynamic way, which is fundamental in functional programming paradigms.

First-Class Function:

This term means that functions in a programming language are treated as first-class citizens. They can be assigned to variables, passed as arguments to other functions, returned as values from other functions, and stored in data structures.

Anonymous functions:

Anonymous functions in JavaScript are functions without a specified name. They are also known as "function literals" or "unnamed functions.”.

Higher-Order Function (HOF):

A Higher-Order Function (HOF) is a type of function in programming that can operate on other functions. This can involve either taking functions as arguments or returning them as results.

Example**:** Consider the function `a(b)`, where `b` is a function itself. In this case, `a` qualifies as a Higher-Order Function (HOF) because it takes another function (`b`) as one of its arguments. This flexibility allows `a` to perform operations on `b` or use it in some way within its own logic.

Callback function:

A callback function is a function that is passed as an argument into another function and is invoked within that function.

Example**:** Function `a` accepts a function `b` as its argument. In this example, `a(b)` is considered a higher-order function (HOF) because it accepts a callback function, `b`, as an argument. Inside `a` function, `b` is invoked using `b()`, making `b` a callback function as it is passed into `a` and executed within it.

Typecasting:

- **Dynamic Typing:** JavaScript is dynamically typed, meaning variable types are determined at runtime.
- **Implicit Typecasting:** JavaScript performs automatic type conversion, also known as type coercion, during operations.
- **Explicit Typecasting:** Developers can explicitly convert types using functions like `parseInt()`, `parseFloat()`, `String()`, `Number()`, etc.
- **Coercion Rules:** JavaScript follows specific rules when coercing types, such as converting non-strings to strings for concatenation.
- **Falsy and Truthy:** JavaScript has truthy and falsy values, impacting conditionals. For example, `0` and `""` are falsy, while non-empty strings and numbers are truthy.
- **NaN:** Represents "Not-a-Number" and is a result of invalid numeric operations.
- **Avoid Implicit Coercion:** It's recommended to explicitly cast types to avoid unexpected results and improve code clarity.
- **Use === for Strict Equality:** To avoid coercion, use `===` for strict equality comparisons.
- **Common Type Issues:** Be cautious with type-related bugs, especially when dealing with different data sources.
- **Type Conversions:** Understand how different operations might implicitly convert types, like adding a string and a number.
- **Constructor Functions:** JavaScript provides constructor functions like `Number()`, `String()`, and `Boolean()` for explicit type conversions.
- **Typeof Operator:** Use the `typeof` operator to check the data type of a variable.
- **Typecasting Arrays:** Arrays can be transformed into strings and back using methods like `join()` and `split()`.

Ways to declare a variable is using let, var, const

How Functions Run Behind the Scenes in JavaScript:

- **Global Execution Context (GEC):**
  - Created when JavaScript code is executed.
  - Comprises two phases: Memory Creation Phase (MCP) and Code Execution Phase (CEP).
- **Memory Creation Phase (MCP):**
  - Occurs before code execution.
  - Allocates memory/space for variables (initializing them with `undefined`) and functions.
  - Prepares the environment.
- **Code Execution Phase (CEP):**
  - Executes code line by line.
  - Assigns values to variables based on code logic.
  - Handles function execution differently:
    - When a function is called, a new Execution Context is created, called Function Execution Context (FEC).
    - FEC also has MCP and CEP.
    - After the function completes, FEC is destroyed.
    - Checks if anything remains to execute in GEC.
    - If no pending tasks, GEC is also destroyed.
- **Call Stack:**
  - GEC and EC are stored and executed in the call stack.
  - The call stack keeps track of the execution flow.
  - Continues running until it becomes empty.

This process outlines how JavaScript functions run behind the scenes, involving the creation of execution contexts and their management in the call stack.

JavaScript Variable Declaration:

- `var` was the traditional way to declare variables in JavaScript before ECMAScript 6 (ES6).
- ES6 introduced two more ways to declare variables: `let` and `const`.

Redeclaration of Variables

- Redeclaration means declaring the same variable name more than once.
- Example of redeclaration:

```
bashCopy code
let a = 10;
let a = 20;
```

Reassignment of Variables

- Reassignment means assigning a different value to a variable that has already been declared.
- Example of reassignment:

```
cssCopy code
let a = 10;
a = 20;
```

- `var` allows redeclaration and reassignment.(They have function-level scope)
- `let` does not allow redeclaration but allows reassignment.(They have block-level scope)
- `const` does not allow redeclaration or reassignment.

Scope:

- In Execution context/Functional Execution context (FEC) `let` and `const` have block scope which is represented by curly brackets `{}` , and `var` has functional or local scope, limited to the function where it's defined.
- In Global Execution Context (GEC) `let` and `const` have script scope, and `var` has global scope.
- In Child Execution Context (CEP):
    - When a variable is accessed, CEP first checks its local memory or memory creation phase(MCP)
    - If the variable is not present in local memory, it checks in its parent's lexical scope.
    - The scope encompasses both local memory and the lexical environment (which is the combination of local memory and the parent's local memory).
    - This process can continue up the parent chain until it reaches the Global Execution Context (GEC) or finds the value of the variable it's searching for.

In summary, scope determines the visibility and accessibility of variables and is influenced by the type of variable declaration (`let`, `const`, or `var`) and the context in which they are used.

Hoisting:

Hoisting is a fundamental concept in JavaScript that allows variables and functions to be accessed even before they are formally declared within the code. This behavior can be understood through the execution of JavaScript code.
But in the case of let and const it’s a little diff it goes to the dead temporal zone and gives a reference error that can’t be accessed before initialization for let and const

- In JavaScript, variables declared using `var` are hoisted. This means that they are moved to the top of their containing function or scope during the compilation phase.
- Variables declared with `var` are initially assigned the value `undefined`. This is why attempting to access a `var` variable before its declaration results in a value of `undefined`.
- `let` and `const` declarations are also hoisted, but with a subtle difference. They are placed in what is referred to as the "Dead Temporal Zone" (DTZ) before they are

initialized.

- In the DTZ, variables declared with `let` and `const` cannot be accessed. Attempting to do so will result in a reference error, indicating that the variable cannot be accessed before initialization.

The Dead Temporal Zone (DTZ) is the state between the initial creation of a `let` or `const` variable and its actual initialization with a value. During this period, trying to access a variable in the DTZ will result in a reference error. Only after initialization can the variable be used.

Closures:

- A closure is a function bundled together with a reference to its surrounding state or lexical environment.
- Essentially, it's a function that "remembers" the variables from the scope in which it was created.
- Closures are created when a function returns another function, and this inner function still has access to the variables of the outer function.
- Closures are commonly used to create private methods, which means not everyone can access them.
- When you try to access a property of an object in JavaScript, the language will first look for it inside that object. If it's not found, it will then search in the object's prototype chain.

***Note:** Closures are a powerful and fundamental concept in JavaScript that allows for data encapsulation and maintaining the integrity of data by limiting access to it. They are often used in scenarios where you want to create encapsulated or private functionality within your code.*

Prototype:

- A prototype in JavaScript is an object that serves as a fallback source of properties for another object.
- When an object is created, it is linked to a prototype object. If a property or method is not found on the object itself, JavaScript looks up the prototype chain to find it.

`__proto__` Property:

`__proto__` (dender proto) is a reference to the prototype of an object. It is used to access the prototype of any object. For example, `Object.prototype` is the parent prototype, and its `__proto__` is `null`. `null` marks the end of the prototype chain, and it's often referred to as the root. The chain of prototypes, from the current object to `null`, is known as the prototypal chain.

Inheritance:

- In JavaScript, inheritance allows objects to access properties and methods from their prototypes.
- For instance, to access properties of a string, JavaScript goes up the prototype chain from the string object to the `Object.prototype` and so on until it reaches `null`.
- This chain of accessing properties and methods through prototypes is what we call the "prototypal chain."

Constructor function:

- A constructor function is a function that is used to create objects.
- When we use the `new` keyword before calling a function, it turns that function into a constructor function.
- The purpose of a constructor function is to create and initialize new objects.
- Inside a constructor function, we can use `this` keyword to refer to the object being created.
- Constructor functions serve as blueprints for creating objects with similar properties and behaviors.
- We can create our own prototypes using constructor functions, which define the shared functionality for objects created from that constructor.

Constructor Function Prototype

- The constructor function prototype is the function itself.

- When you add a function to a constructor function, it is created again for each object created from that constructor.
- To avoid duplicating functions for each object, you can add the function to the constructor function's prototype.
- Adding functions to the prototype allows all objects created from the constructor to share the same function, saving memory and improving efficiency.

Convention- *A convention is a recommended or better practice but not compulsory. In JavaScript, it's a convention to start the name of a constructor function with a capital letter. /For example, if you have a constructor function for creating objects, it's common to name it with an initial capital letter.*

Class syntax:

- A class is a fundamental concept in JavaScript, introduced in ECMAScript 6 (ES6).
- Before classes, developers often used closures to create private functions and encapsulate data.
- Classes provide a more structured and convenient way to work with (define)objects(with properties and methods) and constructors.
- Inheritance allows for the creation of subclasses that inherit features from parent classes.
- The `super` keyword is used to access the constructor and properties of the parent class.

Class Declaration

- To declare a class, use the `class` keyword followed by the class name.
- Conventionally, class names start with a capital letter.
- Inside the class, you define a constructor function using the `constructor` keyword.

```
javascriptCopy code
class MyClass {
  constructor(param1, param2) {
    // Constructor logic here
  }
}
```

Class Members

- Within a class, you can define properties (data members) and methods (functions).
- Properties store data, while methods perform actions.
- You can access properties and methods using `this` keyword within the class.

```
javascriptCopy code
class MyClass {
  constructor(prop1, prop2) {
    this.prop1 = prop1; // Property
    this.prop2 = prop2; // Property
  }

  myMethod() {
    // Method logic here
  }
}
```

Inheritance

- Inheritance allows a new class to inherit properties and methods from an existing class.
- You use the `extends` keyword to create a subclass that inherits from a superclass.
- The `super` keyword is used inside the subclass constructor to call the superclass constructor.

```
javascriptCopy code
class Subclass extends Superclass {
  constructor(prop1, prop2, newProp) {
    super(prop1, prop2); // Call the superclass constructor
    this.newProp = newProp; // Additional property specific to the subclass
  }
}
```

Example Usage

- To create an instance of a class, use the `new` keyword followed by the class name.
- You can pass arguments to the class constructor to initialize its properties.

```
javascriptCopy code
const instance = new MyClass(arg1, arg2);
```

This keyword:

The behavior of the "this" keyword in JavaScript can vary depending on how a function is called. It determines what object or context the function has access to. Here, we'll explore the five common ways in which "this" behaves:

1. Regular Function Invocation

- When a function is invoked directly, "this" refers to the global object.
- In a browser environment, the global object is the "window" object.

Example:

```
javascriptCopy code
function f() {
  // ...
}
f(); // "this" points to the global object (e.g., "window" in the browser)
```

2. Method Invocation

- When a function is invoked as a method of an object, "this" points to the object itself.

Example:

```
javascriptCopy code
let obj = {
  fn: function() {
    // ...
  }
};
obj.fn(); // "this" points to "obj"
```

3. Constructor Invocation

- When a function is used as a constructor with the "new" keyword, "this" refers to the newly created object.

Example:

```
javascriptCopy code
function MyClass() {
  // ...
}
let obj = new MyClass(); // "this" points to the newly created "obj"
```

4. Indirect Calling (call, apply, and bind)

- The "call" and "apply" methods allow you to invoke a function with a specific "this" value.
- "call" and "apply" let you borrow a function from one object and invoke it in the context of another.
- "bind" is used to create a new function with a permanently bound "this" value.

Example:

```
javascriptCopy code
let obj1 = {
  fn: function() {
    // ...
  }
};
let obj2 = {};
obj1.fn.call(obj2); // "this" points to "obj2"
```

- "bind" Example:

```
javascriptCopy code
let obj1 = {
  fn: function() {
    // ...
  }
};
let obj2 = {};
let boundFn = obj1.fn.bind(obj2); // "boundFn" has "this" bound to "obj2"
boundFn(); // "this" points to "obj2"
```

- Polymorphism: Achieved by using "call" and "apply" to reuse a function's properties from one object in another.

Note: In "bind," the function gets called in regular invocation, and "this" points to the specified object.

5. Arrow Functions

-

Async programming:

- `setTimeout()` is a method available in web browsers. We can access it through the global `window` object. It plays a crucial role in handling delays, which is essential for asynchronous programming.
- `setTimeout()` takes two parameters:
  1. A callback function: This is the action you want to execute after a certain delay.
  2. The delay in milliseconds: It specifies the amount of time you want to wait before executing the callback function.
- When you call `setTimeout()`, the browser registers the provided callback function. This means the browser remembers that it needs to execute that function after the specified delay.
- After the delay period is over, the callback function is added to the callback queue. The callback queue is a waiting area for functions that are ready to be executed.

- The event loop, which is a core part of JavaScript's runtime environment, continuously checks the callback queue. When it finds a function in the queue, it moves it to the call stack.
- The call stack is where functions are executed in JavaScript. When the callback function reaches the top of the call stack, it is executed.
- This entire process, from registering the callback to executing it, happens seamlessly within the browser's runtime environment.

In summary, `setTimeout()` is a vital tool for introducing delays and handling asynchronous tasks in web applications. It allows you to schedule actions to occur after a specified time, enhancing the responsiveness and interactivity of web pages.

What is callback queue?

- A callback queue is part of JavaScript's asynchronous execution model. It's used to manage functions (callbacks) that need to be executed after a specific event or operation, such as a timer or an HTTP request, has been completed.
- **Accessing Web APIs:** To interact with external resources, like making network requests or setting timers, JavaScript uses Web APIs. These APIs are provided by the browser or the environment in which JavaScript is running.
- **Storage in Browser:** When you use Web APIs, the browser stores these operations in its internal memory or storage until they are completed. This allows JavaScript to continue executing without waiting for these potentially time-consuming tasks to finish.
- **Callback Queue Entry:** Once a Web API operation is done, the associated callback function is placed in the callback queue. Each callback function in the queue represents an action that needs to be taken, such as handling a response or performing an action after a timeout.
- **Sequential Execution:** The callback queue operates on a first-come, first-served basis. In cases where multiple callbacks are waiting in the queue, they are executed in the order they were added. This ensures that actions are performed sequentially, maintaining the expected flow of your program.

- **Example:** For instance, when you make an asynchronous request to a server using JavaScript, the response-handling function isn't executed immediately. Instead, it's placed in the callback queue. Once the response arrives, it triggers the execution of the associated function, maintaining a non-blocking and responsive user interface.

Event loop:

- The event loop is a mechanism in JavaScript that enables asynchronous behavior.
- It continually checks the call stack for empty space.
- When the stack is empty, it moves tasks from the callback queue to the stack for execution.
- This ensures that JavaScript remains non-blocking and responsive to various events and tasks.

Example:

Consider an example where you have a timer function setTimeout, which schedules a function to run after a specified delay. The event loop ensures that even while waiting for the timer to expire, other parts of your program can continue to run, keeping your application responsive.

Callback nesting:

- Callback nesting refers to the practice of placing callback functions inside other callback functions.
- This approach is commonly used in asynchronous programming scenarios, such as with `setTimeout`.
- **Advantages of Callback Nesting**:
  - **Sequential Execution**: Callback nesting ensures that the second function runs only after the completion of the first one. This sequential execution can be crucial in certain situations.
- **Disadvantages of Callback Nesting**:

- **Pyramid of Doom/callback Hell**: As more functions are nested within each other, the code tends to grow horizontally, leading to what's known as the "pyramid of doom." This makes the code structure harder to read and maintain.
- **Decency**: The dependency between functions becomes complex. If the first function doesn't execute, subsequent functions won't execute either, which can be challenging to manage.

Promises:

Promises were introduced to simplify the handling of asynchronous operations and mitigate the issues associated with callback hell.

Syntax:

To create a promise, you use the `new Promise()` constructor. The `new` keyword creates an object, but when used with the `Promise` constructor, it returns a promise instead of a regular object. The constructor accepts one function, which is executed immediately(to avoid that we use setTimeOut). This function, in turn, receives two arguments: `resolve` and `reject`, which are functions themselves.

Usage:

- **Resolve:** The `resolve` function is used within the promise when the asynchronous operation is successful. You can pass data as an argument to `resolve`. This data will be handled by the `.then()` function, allowing you to work with the successful result of the asynchronous operation.

  Example:

  ```
  javascriptCopy code
  let p = new Promise((resolve, reject) => {
    // Simulate an asynchronous operation
    setTimeout(() => {
      let data = "Success!";
      resolve(data); // Data is sent to .then()
    }, 1000);
  });

  p.then((result) => {
    console.log(result); // Output: "Success!"
  ```

```
});
```

- **Reject:** Conversely, the `reject` function is used to handle scenarios where the asynchronous operation fails. It can be thought of as the error handler for the promise. Errors raised in the promise can be caught and managed using the `.catch()` method.

  Example:

  ```
  javascriptCopy code
  let p = new Promise((resolve, reject) => {
    // Simulate a failed asynchronous operation
    setTimeout(() => {
      let error = new Error("Something went wrong!");
      reject(error); // Error is sent to .catch()
    }, 1000);
  });

  p.catch((error) => {
    console.error(error.message); // Output: "Something went wrong!"
  });
  ```

Conclusion:

Promises help avoid callback hell by allowing you to handle success and error scenarios with the `resolve` and `reject` functions. The use of `.then()` and `.catch()` methods simplifies the handling of asynchronous code, making it more maintainable and less error-prone.

In this scenario, the code tends to grow vertically, resulting in a series of .then() method calls that lead to what is commonly known as "promise chaining." To mitigate this issue and enhance code readability, the solution is to implement asynchronous programming using `async/await` syntax.

Async await:

- To create an asynchronous function, use the `async` keyword before the function declaration. This marks the function as asynchronous and ensures it returns a promise.

- Within the async function, promises are used to represent asynchronous operations. These promises can be resolved or rejected.
- Async and await are used together. Await is placed before a promise, indicating that the program should wait for the promise to settle (resolve or reject) before proceeding.

Example:

```
javascriptCopy code
async function fetchData() { // Declaring an async function
  try {
    const data = await fetch('https://api.example.com/data'); // Using await with a promis
e
    console.log(data); // Process the data once the promise is resolved
  } catch (error) {
    console.error(error); // Handle errors if the promise is rejected
  }
}
```

- In this example, `fetchData` is an async function.
- `await fetch('https://api.example.com/data')` waits for the data to be fetched from the URL.
- When the promise is resolved, the data is logged.
- If there's an error (the promise is rejected), it's caught in the `catch` block for error handling.
- Async/await simplifies asynchronous code by making it appear more like synchronous code, improving readability and maintainability.

Dom:

- **Document Object Model (DOM)** is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content.
-

DOM Tree:

- The browser provides an object called **"document,"** which represents the HTML page. It creates a **DOM tree** structure. The DOM tree consists of HTML elements, attributes, and their relationships.

Selecting Elements:

1. **getElementById:**
    - Returns an object for a given element ID.
    - If multiple elements have the same ID, only the first one is returned.
2. **getElementsByTagName:**
    - Returns an array or HTMLCollection of elements with the specified tag name.
    - Multiple elements can be selected.
3. **getElementsByClass:**
    - Returns an HTML collection of elements with the specified class name.
    - Multiple elements can be selected.
4. **querySelector:**
    - Can select elements, classes, or IDs using CSS selectors.
    - Returns a single element.
5. **querySelectorAll:**
    - Similar to querySelector but returns multiple elements in the form of an array or nodeList.

Changing Style Properties:

- To change style properties of elements, the **.style** property is used.
- Style properties include **.style.color**, **.style.backgroundColor**, **.style.fontSize**, **.style.fontWeight**, **.style.border**, etc.

Common properties for getting and setting data/text include:

1. **innerText:** Retrieves the text inside an element while considering CSS styles.
2. **textContent:** Retrieves the text inside an element without considering CSS styles. It displays hidden text.

3. **innerHTML:** Returns the complete content of an element, including its tags, text, and style.

Note-When working with HTML collections (e.g., from getElementsByTagName or getElementsByClass), you cannot use push and pop methods. Properties of HTML elements can act as getters and setters.

Attribute Manipulation:

- **Attributes** are additional pieces of information associated with HTML elements that are sent to the browser.
- To **retrieve** or **fetch** the value or text of an attribute, we use the `getAttribute` method.
- To **set** or **modify** attributes, we use the `setAttribute` method, which takes two arguments:
    1. The **name** of the attribute we want to set.
    2. The **value** we want to assign to that attribute.
- There are two types of timing functions in JavaScript for managing time-related tasks:
    - `setTimeout`: Delays the execution of a function or code block after a specified delay.
    - `setInterval`: Repeatedly executes a function or code block at a specified interval.

Working with ClassList:

- **ClassList** is used to manipulate the classes of HTML elements, allowing you to add or remove classes and check if an element has a particular class.
- ClassList provides four methods for working with classes:
    1. `add`: Adds one or more classes to an element.
    2. `remove`: Removes one or more classes from an element.
    3. `toggle`: Toggles the presence of a class on an element. If the class is present, it removes it; if it's not present, it adds it.
    4. `contains`: Checks if an element has a specific class and returns `true` or `false`.

- Example:

  ```
  let element = document.querySelector("a");
  element.classList.add("classname"); // Adds the class "classname" to the element's cl
  ass list.
  ```

In summary, attribute manipulation involves working with HTML element attributes using `getAttribute` and `setAttribute`, while ClassList is a set of methods for manipulating classes on HTML elements, such as adding, removing, toggling, or checking for the presence of classes.

Traversing a DOM Tree:

- **Parent Element:** To navigate to the parent element, use the `.parentElement` property.
- **Child Elements:** For child elements, employ the `.children` property. If there are multiple child elements, use a loop to access and manipulate them.
- **Siblings:** To access the sibling element that comes next, use the `.nextElementSibling` property. For the previous sibling, utilize `.previousElementSibling`.

Adding Elements to the DOM:

- **Creating Elements:** To add an element, first ensure it either exists or create it using the `.createElement()` method.
- **Appending Child Elements:** Use the `.appendChild()` method to add the created child element to its parent. Note that this method only accepts one argument and cannot be used to add text directly.
- **Adding Text:** To add text to a new child element, you can use properties like `innerText` or `textContent`.
- **Adding Multiple Elements:** The `.append()` method can add multiple child elements at once. It appends elements to the end and allows you to add text content as well.
- **Adding to the Beginning:** To add elements at the beginning, utilize the `.prepend()` method. This method also accepts text content.

Removing Elements from the DOM:

- **Using `removeChild()`:** Employ the `.removeChild()` method to remove a child element. This requires specifying both the parent and child elements, like `parentElement.removeChild(childElement)`.
- **Using `remove()`:** To remove an element without referencing the parent, use the `.remove()` method. Simply call `element.remove()` to delete the element.

Note: Console log converts the object into a string and prints it. However, the `console.dir` method displays all the details of that object, providing a more comprehensive view of its properties and structure.

Event Handling:

- Event handling in JavaScript allows us to respond to DOM events and execute specific actions when these events occur.
- Using the `onclick` attribute like `Btn.onclick = function()`, we can assign a single function to be executed when an event happens. However, this has limitations as only one function can be called at a time.

Using `addEventListener`:

- To overcome the limitation of calling only one function for an event, we can use the `addEventListener` method. It accepts two arguments: the event and the function to be executed.
- Syntax: `Btn.addEventListener("click", function1);`
- We can also attach multiple event listeners to the same element, allowing us to call multiple functions for the same event.
- Syntax for multiple event listeners:

```
arduinoCopy code
Btn.addEventListener("click", function1);
Btn.addEventListener("click", function2);
```

Input Events:

- Input events are triggered when the value of an input element changes.
- We can use the `input` event to respond to changes in the input's value.

- Example: `Input.addEventListener("input", function());`

Event Object:

- Whenever an event is executed, an event object is automatically created and can be accessed within the event handler. This object is often named `e` or `event`.

Event Target:

- The event target is the HTML element on which the event is triggered. It is accessible through the event object.
- Example: `event.target` gives us access to the element that triggered the event.

Preventing Default Behavior:

- In some cases, events have default behaviors associated with them. For instance, clicking a link navigates to a new page by default.
- `event.preventDefault()` is used to prevent the default behavior of an event. For example, it can be used to stop a link from navigating to a new page.

Accessing Elements Inside a Form:

- The `elements` property can be used to access elements inside a form. It returns a collection of all form elements.
- Example: `formSelector.elements` gives access to the elements within the form referenced by `formSelector`.