JavaScript
Reading Material

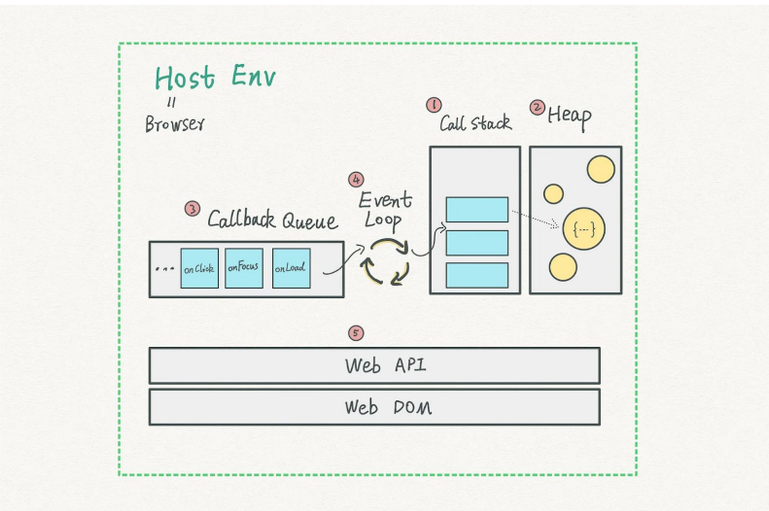
PW SKILLS

Topics

- **Understanding the concept of JavaScript**
  - How Javascript works
  - Variables and Data Types
  - operators
  - Functions, Loops, Conditional Statement
  - Scope
  - Hoisting
    - How it work
    - Temporal dead zone
  - Closures
  - Array Methods and Object Methods
  - Destructuring - Spread and Rest
  - DOM Manipulation
  - Browser Object Model (BOM)
  - Event Handling
  - Object-Oriented Programming in JavaScript
  - Prototype and Prototype Inheritance
  - ES6+ Features
  - The 'this' keyword
  - calls(), apply(), bind()
  - Error Handling (try...catch, types of error)
  - Asynchronous Programming (Callback, Promise, Async-Await)
  - Polyfills
  - Regex
- **Interview Questions**
- **Multiple Choice Questions**

Understanding the concept of JavaScript

How Javascript works



- **JavaScript Engine:** JavaScript runs inside a special program called a JavaScript engine, found in web browsers (like Chrome's V8 engine) or on servers (like Node.js).
- **Code Loading:** When you load a webpage, the browser gets the HTML, CSS, and JavaScript files. It reads (or parses) the JavaScript code and prepares to execute it.
- **Compilation and Execution:** Modern JavaScript engines use a technique called Just-In-Time (JIT) compilation, which means they quickly convert the JavaScript code into machine code that your computer can understand, just before running it.

- **Call Stack:** JavaScript keeps track of what it's doing with a call stack. When it needs to execute a function, it adds (or pushes) it to the stack. When it's done, it removes (or pops) it from the stack.
- **Memory Management:** JavaScript automatically manages memory for you. It keeps track of what data is needed and gets rid of (or garbage collects) data that is no longer used.
- **Event Loop and Asynchronous Operations:** JavaScript can handle multiple things at once using an event loop. Here's how it works:
  - Call Stack: Manages function calls.
  - Web APIs: Browser features like setTimeout or fetch handle tasks in the background.
  - Task Queue: Holds tasks to be run when the call stack is clear.
- **Event Loop:** Moves tasks from the task queue to the call stack when it's empty.

Variables and Data Types

- 1. Declaration Keywords:
  - **var:** Declares a variable, optionally initializing it to a value. It's function-scoped and can be re-declared and updated.
  - **let:** Declares a block-scoped variable, optionally initializing it to a value. It can be updated but not re-declared within the same scope.
  - **const:** Declares a block-scoped, read-only constant. The value must be initialized during the declaration and cannot be re-assigned.

```
var x = 10;
let y = 20;
const z = 30;
```

Data Types in JavaScript

- **1. Primitive Data Types:**
  - Number: Represents both integer and floating-point numbers.
  - String: Represents a sequence of characters.
  - Boolean: Represents a logical entity with two values: true or false.
  - Undefined: A variable that has been declared but not initialized.
  - Null: Represents the intentional absence of any object value.
  - Symbol: Represents a unique and immutable primitive value.
- **2. Object Data Types:**
  - Object: A collection of key-value pairs.
  - Array: A list-like object used to store multiple values.
- **3. Special Data Type:**
- Function: Represents a block of code designed to perform a particular task.

```
//Number
let num = 42;
let price = 19.99;
console.log(typeof 42); // "number"

//String
let name = "Alice";
console.log(typeof name ); // "string"

//Boolean
let isAvailable = true;
console.log(typeof isAvailable); // "boolean"
```

```
//Undefined
let x;
console.log(x); // undefined
console.log(typeof x); // "undefined"

//Null
let y = null;
console.log(typeof y ); // "object" (this is a quirk in
JavaScript)


//Symbol
let sym = Symbol("unique");
console.log(typeof Symbol(sym)); // "symbol"

//Object
let person = {
  name: "Alice",
  age: 30
};
console.log(typeof {name: person }); // "object"


//Array
let numbers = [1, 2, 3, 4, 5];
console.log(typeof numbers ); // "object" (arrays are objects in
JS)

//function
function greet() {
  console.log("Hello, world!");
}
console.log(typeof function() {}); // "function"
```

Operators

expressions are combinations of values, variables, operators, and functions that are interpreted and evaluated to produce a result. Operators are symbols that perform operations on operands, which can be values, variables, or expressions. There are several types of operators in JavaScript:

- **Arithmetic Operators:** Used to perform arithmetic operations on numerical values.
- **Assignment Operators:** Used to assign values to variables.
- **Comparison Operators:** Used to compare two values and return a Boolean result.
- **Ternary Operator (Conditional Operator):** Used to assign a value to a variable based on a condition.
- **Logical Operators:** Used to combine conditional statements.

```
//Arithmetic Operators
let sum = a + b;           // Addition
let difference = a - b;    // Subtraction
let product = a * b;       // Multiplication
let quotient = a / b;      // Division
let remainder = a % b;     // Modulus
let power = a ** b;        // Exponentiation
//Assignment Operators
```

```
let x = 10;        // Assignment
x += 5;            // Addition assignment (x = x + 5)
x -= 3;            // Subtraction assignment (x = x - 3)
x *= 2;            // Multiplication assignment (x = x * 2)
x /= 2;            // Division assignment (x = x / 2)
x %= 3;            // Modulus assignment (x = x % 3)
x **= 2;           // Exponentiation assignment (x = x ** 2)
// Comparison Operators
let isEqual = a == b;            // Equal to
let isStrictEqual = a === b;     // Strictly equal to
let isNotEqual = a != b;         // Not equal to
let isStrictNotEqual = a !== b; // Strictly not equal to
let isGreaterThan = a > b;       // Greater than
let isLessThan = a < b;          // Less than
let isGreaterOrEqual = a >= b;  // Greater than or equal to
let isLessOrEqual = a <= b;      // Less than or equal to
//Ternary Operator (Conditional Operator)
let result = condition ? valueIfTrue : valueIfFalse;
//Logical Operators
let andResult = a && b;    // Logical AND
let orResult = a || b;     // Logical OR
let notResult = !a;        // Logical NOT
```

Functions, Loops, Conditional Statement

Function: A function is a reusable block of code designed to perform a particular task or calculate a value. Functions are fundamental building blocks in JavaScript, allowing you to organize your code into modular, manageable pieces.

Types of functions:

- **Function Declaration:** A named function defined with the function keyword.
- **Function Expression:** A function defined within an expression, can be anonymous or named.
- **Arrow Function:** A concise function expression using the => syntax, with no own this context.
- **Anonymous Function:** A function without a name, often used as a callback or for one-time use where a function declaration isn't needed.
- **Constructor Function:** A function used to create objects with the new keyword.
- **Generator Function:** A function using function* syntax that can pause and resume its execution with yield.
- **Async Function:** A function that returns a Promise and allows await for asynchronous operations.
- **IIFE (Immediately Invoked Function Expression):** It's a function expression that's immediately invoked after its creation, often used to create a private scope.
- **HOF (Higher-Order Function):**A function that either takes a function as an argument or returns a function, enabling functional programming paradigms.

```
function namedFunction() {
    return "I'm a function Declaration";
}

//Function Expression
const functionExpression = function() {
    return "I'm a function expression";
};
```

```
//Arrow Function
const arrowFunction = () => {
    return "I'm an arrow function";
};
//Anonymous Function
const anonymousFunction = function() {
    return "I'm an anonymous function";
};

//Generator Function
function* generatorFunction() {
    yield "I'm a generator function";
}

//Async Function
async function asyncFunction() {
    return "I'm an async function";
}

//Immediately Invoked Function Expression (IIFE):
(function() {
    console.log("I'm an IIFE");
})();

//Higher-Order Function
function higherOrderFunction(callback) {
    return callback();
}
Loop: Loops are powerful tools for performing repetitive tasks
efficiently. Loops in JavaScript execute a block of code again and
again while the condition is true.
Different types of loop in Javascript
//1.for Loop - execute block of code repeatedly until specified
condition is met.
for (initialization; condition; iteration) {
    // code to be executed
}

/* 2.while Loop: executes a block of code repeatedly, then checks
specified    condition after earch execution */
while (condition) {
    // code to be executed
}

/* 3.do...while Loop:execute block of code repeatedly, then checks
a specified condition after each execution */
do {
```

```
    // code to be executed
} while (condition);

//for...in Loop (used for iterating over the properties of an
object):
for (variable in object) {
    // code to be executed
}

//4.for...of Loop (used for iterating over iterable objects like
arrays, strings, etc.):
for (variable of iterable) {
    // code to be executed
}
```

Conditional Statement: A condition statement is a control structure that evaluates whether a given condition is true or false. Based on the result of this evaluation, different blocks of code are executed. Here are the main types of condition statements in JavaScript:
Different types of condition statements

```
//1. if Statement: executes a block of code if a specified
condition is true.
if (condition) {
    // code to be executed if the condition is true
}

/* 2.if...else Statement: executes one block of code if the
condition is true and another block of code if the condition is
false.
*/
if (condition) {
    // code to be executed if the condition is true
} else {
    // code to be executed if the condition is false
}

/* 3.if...else if...else Statement:  check multiple conditions and
execute different blocks of code based on which condition is true.
*/
/*4. switch Statement: evaluates an expression and executes a
block of code depending on a matching case value. */
switch (expression) {
    case value1:
        // code to be executed if expression equals value1
        break;
    case value2:
        // code to be executed if expression equals value2
        break;
    default:
        // code to be executed if expression doesn't match any
case
}
```

Scope

scope refers to the accessibility and visibility of variables, functions, and objects within different parts of a codebase. There are two main types of scope in JavaScript: global scope and local scope.

- **Global Scope:** Variables declared outside of any function or block have a global scope. They can be accessed and modified from any part of the code, including within functions.
- **Local Scope:** Variables declared within a function have local scope, meaning they are only accessible within that function.

Block Scope: Block scope refers to the scope of variables declared inside a block, which is a pair of curly braces {}. Variables declared with let or const inside a block are limited to that block.

```
/* 1. Global Scope: Variables declared outside of any function or
block have a global scope. They can be accessed and modified from
any part of the code, including within functions.
*/
const globalVariable = "I'm a global variable";
function testFunction() {
    console.log(globalVariable); // Accessible
}
/*2.Local Scope: Variables declared within a function have local
scope, meaning they are only accessible within that function.*/
function testFunction() {
    var localVariable = "I'm a local variable";

 console.log(localVariable); // Accessible
}
console.log(localVariable); // Error: localVariable is not defined

/*3.Block Scope: refers to the scope of variables declared inside
a block, which is a pair of curly braces {}. Variables declared
with let or const inside a block are limited to that block.*/
if (true) {
    let blockScopedVariable = "I'm a block-scoped variable";
    console.log(blockScopedVariable); // Accessible
}
console.log(blockScopedVariable); // Error: blockScopedVariable is
not defined
```

Hoisting

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their containing scope during the compile phase. This means that you can use variables and functions before you declare them in your code.

How Hoisting Works:

1. Variables declared with var:

- When you declare a variable using var, JavaScript "hoists" the declaration to the top of the function or global scope.
- Only the declaration is hoisted, not the initialization.
- This means the variable is undefined until the line where it's initialized.

```
console.log(x); // undefined
var x = 5;
console.log(x); // 5
Behind the scenes, it's like this:
var x;
console.log(x); // undefined
x = 5;
console.log(x); // 5
```

2. Variables declared with let and const:
- let and const are also hoisted, but they are not initialized.

They are placed in a "temporal dead zone" from the start of the block until the declaration is encountered, meaning you cannot access them before the declaration line.

```
console.log(y); // ReferenceError: Cannot access 'y' before
initialization
let y = 10;

console.log(z); // ReferenceError: Cannot access 'z' before
initialization
const z = 20;
```

3. Function Declarations:
- Function declarations are hoisted with both their name and the function body.
- This means you can call a function before you declare it.

```
greet(); // "Hello!"
function greet() {
    console.log("Hello!");
}
```

Understanding hoisting helps avoid unexpected behavior and bugs related to variable and function usage in your JavaScript programs

Temporal dead zone:

The Temporal Dead Zone (TDZ) in JavaScript is a period during which variables declared with let and const exist but cannot be accessed. This period starts from the beginning of the block until the variable's declaration is encountered. Attempting to access variables within the TDZ results in a ReferenceError. The TDZ helps prevent accidental use of uninitialized variables and enforces better coding practices.

```
{
    console.log(a); // ReferenceError: Cannot access 'a' before
initialization
    let a = 10;
    console.log(a); // 10
}
```

Closures

Before closures, let's take a look at the lexical scope.
Lexical scope is a concept in JavaScript that determines the availability of variables and identifiers in the code based on where they are defined in the source code. In other words, it defines the visibility of variables and functions within nested functions.
A closure is the combination of a function and the lexical environment within which that function was declared. It allows a function to access and manipulate variables from its outer scope, even after the outer scope has finished executing.

Before closures, let's take a look at the lexical scope.
Lexical scope is a concept in JavaScript that determines the availability of variables and identifiers in the code based on where they are defined in the source code. In other words, it defines the visibility of variables and functions within nested functions.
A closure is the combination of a function and the lexical environment within which that function was declared. It allows a function to access and manipulate variables from its outer scope, even after the outer scope has finished executing.

```
function outerFunction() {
    let outerVar = 'I am from outerFunction';

    function innerFunction() {
        console.log(outerVar); // Accessing outerVar from
outerFunction
    }

    return innerFunction;
}
let innerFunc = outerFunction();
innerFunc(); // Outputs: "I am from outerFunction"
```

Array Methods and Object Methods

Here's a list of commonly used array methods and object methods in JavaScript

```
/* 1. push(): Adds one or more elements to the end of an array and
returns the new length */
Syntax: array.push(element1, ..., elementN)
let arr = [1, 2, 3];
arr.push(4); // arr is now [1, 2, 3, 4]

/*2. pop():Removes the last element from an array and returns that
element*/
Syntax: array.pop()
let arr = [1, 2, 3];
arr.pop(); // arr is now [1, 2]

/*3. shift():Removes the first element from an array and returns
that element, shifting all other elements to a lower index */
Syntax: array.shift()
let arr = [1, 2, 3];
arr.shift(); // arr is now [2, 3]

/*4. unshift():Adds one or more elements to the beginning of an
array and returns the new length.*/
Syntax: array.unshift(element1, ..., elementN)
let arr = [1, 2, 3];
arr.unshift(0); // arr is now [0, 1, 2, 3]
```

```
//5.concat():Combines two or more arrays and returns a new array.
Syntax: array1.concat(array2, ..., arrayN)
let arr1 = [1, 2];
let arr2 = [3, 4];
let result = arr1.concat(arr2); // result is [1, 2, 3, 4]

/*6.slice():Returns a shallow copy of a portion of an array into a
new array object selected from begin to end (end not included).*/
Syntax: array.slice(begin, end)
let arr = [1, 2, 3, 4];
let result = arr.slice(1, 3); // result is [2, 3]

/* 7.splice():Changes the contents of an array by removing or
replacing existing elements and/or adding new elements.*/
Syntax: array.splice(start, deleteCount, item1, ..., itemN)
let arr = [1, 2, 3, 4];
arr.splice(1, 2, 'a', 'b'); // arr is now [1, 'a', 'b', 4]
//8.map(): map throught the items array and return new array.
map(): Creates a new array with the results of calling a provided
function on every element in the calling array.

//9.forEach():Calls a function once for each element in the array.
Syntax: array.forEach(function(currentValue, index, array))
let arr = [1, 2, 3];
arr.forEach(element => console.log(element)); // logs 1, 2, 3

//10.map():Creates a new array with the results of calling a
provided function on every element in the calling array.
Syntax: array.map(function(currentValue, index, array))
let arr = [1, 2, 3];
let result = arr.map(x => x * 2); // result is [2, 4, 6]

/*11.filter():Creates a new array with all elements that pass the
test implemented by the provided function.*/
Syntax: array.filter(function(currentValue, index, array))
let arr = [1, 2, 3, 4];
let result = arr.filter(x => x > 2); // result is [3, 4]

/*12.find():Returns the value of the first element in the array
that satisfies the provided testing function.*/
Syntax: array.find(function(element, index, array))
let arr = [1, 2, 3, 4];
let result = arr.find(x => x > 2); // result is 3

/*13.findIndex():Returns the index of the first element in the
array that satisfies the provided testing function.*/
Syntax: array.findIndex(function(element, index, array))
let arr = [1, 2, 3, 4];
let result = arr.findIndex(x => x > 2); // result is 2
```

```
/*14.indexOf(): return the index of the specified array item*/
let arr = [1, 2, 3, 4];
let result = arr.indexOf(3); // result is 2

/*15.includes():Determines whether an array includes a certain
element, returning true or false as appropriate.*/
Syntax: array.includes(searchElement, fromIndex)
let arr = [1, 2, 3];
let result = arr.includes(2); // result is true

/* 16. reduce():Applies a function against an accumulator and each
element in the array (from left to right) to reduce it to a single
value.*/
Syntax: array.reduce(function(accumulator, currentValue, index,
array), initialValue)
let arr = [1, 2, 3, 4];
let result = arr.reduce((sum, current) => sum + current, 0); //
result is 10
```

Object Methods:

```
/*1.Object.keys(): Returns an array of a given object's own
enumerable property names.*/
Syntax: Object.keys(obj)
let obj = { a: 1, b: 2 };
let keys = Object.keys(obj); // keys is ['a', 'b']

/*2. Object.values(): Returns an array of a given object's own
enumerable property values.*/
Syntax: Object.values(obj)
let obj = { a: 1, b: 2 };
let values = Object.values(obj); // values is [1, 2]

/*3.Object.entries(): Returns an array of a given object's own
enumerable string-keyed property [key, value] pairs.*/
Syntax: Object.entries(obj)
let obj = { a: 1, b: 2 };
let entries = Object.entries(obj); // entries is [['a', 1], ['b',
2]]

/*4. Object.assign(): Copies the values of all enumerable own
properties from one or more source objects to a target object.*/
Syntax: Object.assign(target, ...sources)
let target = { a: 1 };
let source = { b: 2 };
let returnedTarget = Object.assign(target, source); // target is {
a: 1, b: 2 }

/*5.Object.hasOwnProperty(): Returns a boolean indicating whether
the object has the specified property as its own property (not
inherited).*/
Syntax: obj.hasOwnProperty(prop)
let obj = { a: 1 };
let result = obj.hasOwnProperty('a'); // result is true
```

```
//6.Object.freeze(): Freezes an object: other code cannot delete
or change any properties.
Syntax: Object.freeze(obj)
let obj = { a: 1 };
Object.freeze(obj);
obj.a = 2; // Does nothing in strict mode, throws error in non-
strict mode

//7.Object.seal():Prevents new properties from being added to an
object and marks all existing properties as non-configurable.
Syntax: Object.seal(obj)
let obj = { a: 1 };
Object.seal(obj);
obj.b = 2; // Does nothing

//8.Object.create():Creates a new object with the specified
prototype object and properties.
Syntax: Object.create(proto, [propertiesObject])
let proto = { a: 1 };
let obj = Object.create(proto);
console.log(obj.a); // 1
```

Destructuring - Spread and Rest

Destructuring allows you to extract values from arrays or properties from objects and assign them to variables in a concise and readable way.
Array Destructuring:

```
const numbers = [1, 2, 3];
const [a, b, c] = numbers;
console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
```

Object Destructuring:

```
const person = { name: 'John', age: 30 };
const { name, age } = person;
console.log(name); // 'John'
console.log(age); // 30
```

Rest Operator (...):
The rest operator allows you to represent an indefinite number of arguments as an array in a function declaration. It's useful when you want a function to accept any number of arguments without explicitly defining them.

```
//Syntax:
function functionName(...args) {
    // 'args' is an array containing all passed arguments
}

function sum(...numbers) {
    return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3)); // Output: 6
console.log(sum(1, 2, 3, 4, 5)); // Output: 15
```

Spread Operator (...):
The spread operator allows an iterable (like an array) to be expanded into individual elements. It's useful for functions that require separate arguments or for creating copies of arrays or objects.

```
//Syntax:
function functionName(...args) {
    // 'args' is an array containing all passed arguments
}

function sum(...numbers) {
    return numbers.reduce((total, num) => total + num, 0);
}
console.log(sum(1, 2, 3)); // Output: 6
console.log(sum(1, 2, 3, 4, 5)); // Output: 15
```

DOM Manipulation

DOM (Document Object Model) manipulation involves using JavaScript to interact with and modify HTML and CSS on a webpage. It allows dynamic changes to the content, structure, and style of a webpage.
Key Concepts:

- **Selecting Elements:** Identifying the elements you want to manipulate.
- **Modifying Elements:** Changing the content, attributes, or styles of elements.
- **Creating and Removing Elements:** Adding or removing elements from the DOM.

Selecting Elements:

- **getElementById:** Selects a single element by its ID.
- **getElementsByClassName:** Selects all elements with a specific class.
- **getElementsByTagName:** Selects all elements with a specific tag name.
- **querySelector:** Selects the first element that matches a CSS selector.
- **querySelectorAll:** Selects all elements that match a CSS selector.

```
//getElementById
const element = document.getElementById('myElement');

//getElementsByClassName
const elements = document.getElementsByClassName('myClass');

//getElementsByTagName
const elements = document.getElementsByTagName('div');

//querySelector
const element = document.querySelector('.myClass');

//querySelectorAll
const elements = document.querySelectorAll('.myClass');
```

Modifying Elements:

- **Change Content:** Use innerHTML or textContent to change the content.
- **Change Attributes:** Use setAttribute, getAttribute, and removeAttribute.
- **Change Styles:** Use the style property or classList to modify styles.

Creating and Removing Elements:

- **Create Elements:** Use createElement and appendChild.
- **Remove Elements:** Use removeChild or remove.

```
element.setAttribute('src', 'image.jpg');
const src = element.getAttribute('src');
element.removeAttribute('src');
```

Browser Object Model (BOM)

The Browser Object Model (BOM) allows JavaScript to interact with the browser outside of the document content. It provides various objects to interact with the browser window and control things like the browser history, screen, and navigation.
Key Components of the BOM
Window Object: The window object represents the browser window and is the global object in JavaScript running in the browser. All global JavaScript objects, functions, and variables automatically become members of the window object.
Navigator Object: The navigator object contains information about the browser and the user's environment.
Screen Object : The screen object contains information about the user's screen.
History Object: The history object provides an interface for manipulating the browser session history (pages visited in the tab or frame).
Location Object: The location object contains information about the current URL and provides methods to manipulate it.
Timing event: Timing events in the window object allow for delayed or repeated execution of functions.
dialog box methods: Interact with users through dialog boxes for displaying messages, obtaining confirmation, and prompting for input.

```
//window controll 
window.open() // open window
window.close() //close window
window.moveTo() //move to ".." window
window.resizeTo() // resixe the window 

// navigation
navigator.cookieEnabled //return true if cookie are enable
navigator.appName //application name of the browser
navigator.online //return true of browser it online
navigator.platform //return the operating systum
navigator.language //return the language of the browser

//screen
screen.width
screen.height
screen.availwidth
screen.availHeight
screen.colorDepth
screen.pixelDepth
```

```
//history
window.history.back() // go back
window.history.forwork() // go forword

//location
window.location.href // href (URL) of the current page
window.location.hostname // domain name of web host
window.location.pathname //path and filename of the current  page
window.location.protocal //the web protocol used(http: or https:)
window.location.assign() //loads a new document

// timing event
can be written without window
window.setTimeout()
setTimeout()
setInterval()
clearInterval()
clearTimeout()

//popups or dialog box methods
can be written without window
window.alert()
alert()
confirm()
prompt()
```

Event Handling

Event handling in JavaScript is a fundamental concept that allows you to interact with users and respond to their actions, such as clicks, key presses, mouse movements, and more. Here's a comprehensive overview of event handling in JavaScript:

Events: Actions or occurrences that happen in the browser, such as clicks, key presses, or mouse movements.
Event Listeners: Functions that are called when an event is fired.
Adding Event Listeners :

- **addEventListener Method:** This method attaches an event handler to an element without overwriting existing event handlers.

```
//Syntax:
element.addEventListener(event, handler, [options]);
document.getElementById('myButton').addEventListener('click',
function() {
    alert('Button clicked!');
});
```

Parameters:

- **event:** The name of the event (e.g., 'click', 'mouseover').
- **handler:** The function to handle the event.
- **options (optional):** An object that specifies characteristics about the event listener.

Removing Event Listeners
removeEventListener Method: This method removes an event handler that was attached with addEventListener.

```
//Syntax:
element.removeEventListener(event, handler, [options]);

function handleClick() {
    alert('Button clicked!');
}

const button = document.getElementById('myButton');
button.addEventListener('click', handleClick);
button.removeEventListener('click', handleClick);
```

Event Object:

- Contains details about the event (e.g., event.target, event.type).
- Methods like event.preventDefault() and event.stopPropagation()

```
document.getElementById('myButton').addEventListener('click',
function(event) {
    console.log(event.target); // The button element
    console.log(event.type); // 'click'
    event.preventDefault(); // Prevent the default action
});
```

Event Propagation

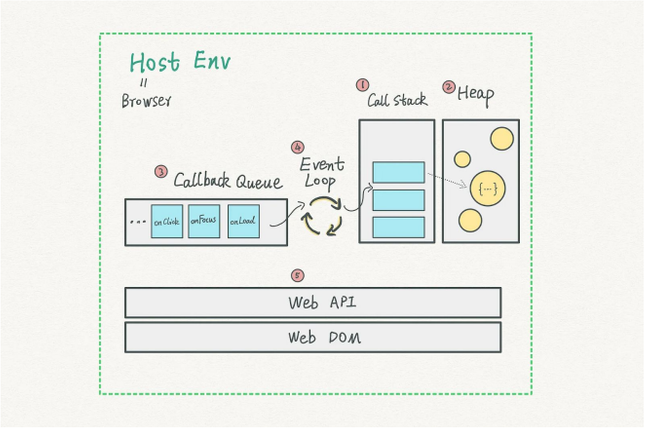
Event propagation describes how events flow through the Document Object Model (DOM) tree. There are three phases:

- **Capturing Phase:** The event travels from the root to the target element. Listeners registered for this phase will handle the event as it travels down.

```
element.addEventListener('click', handler, true); // true for
capturing
```

- **Target Phase:** The event reaches the target element. Listeners registered for this phase will handle the event when it directly affects the target element.
- **Bubbling Phase:** The event bubbles up from the target element to the root. Listeners registered for this phase will handle the event as it travels back up.

```
element.addEventListener('click', handler); // default is false,
for bubbling
```



Stopping Propagation

- stopPropagation(): Stops the event from propagating further (both capturing and bubbling).

```
event.stopPropagation();
```

- stopImmediatePropagation(): Stops the event from propagating and prevents other listeners of the same event from being called.

```
event.stopImmediatePropagation();
```

Event Delegation
Event Delegation is a technique where a single event listener is added to a parent element to manage events for all of its child elements, instead of adding separate event listeners to each child. This technique is efficient and reduces memory usage, especially when dealing with many child elements.

Object-Oriented Programming in JavaScript

Object-Oriented Programming (OOP) is a programming paradigm that uses objects to design and structure software. JavaScript supports OOP through its prototype-based inheritance model.

Class Declaration : A class in JavaScript is declared using the class keyword followed by the name of the class. Here's the basic syntax:

```
class ClassName {
  constructor(parameter1, parameter2) {
    // Initialize properties
    this.property1 = parameter1;
    this.property2 = parameter2;
  }
  // Method
  method1() {
    console.log('This is a method');
  }
  // Another method
  method2() {
    console.log('This is another method');
  }
}
```

Class Expressions : Classes can also be defined using class expressions. Class expressions can be named or unnamed. Named class expressions are not hoisted, meaning you cannot use them before they are declared.

```
// Unnamed class expression
const MyClass = class {
  constructor() {
    console.log('Unnamed class expression');
  }
};

// Named class expression
const AnotherClass = class Another {
  constructor() {
    console.log('Named class expression');
  }
};
```

Inheritance: Classes can extend other classes using the extends keyword. This allows the creation of a class hierarchy.

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

  speak() {
    console.log(`${this.name} makes a noise.`);
  }
}

class Dog extends Animal {
  constructor(name, breed) {
    super(name); // Call the parent class's constructor
    this.breed = breed;
  }

  speak() {
    console.log(`${this.name} barks.`);
  }
}

const dog = new Dog('Rex', 'German Shepherd');
dog.speak(); // Output: Rex barks.
```

Static Methods: Static methods are defined on the class itself, not on instances of the class. They are typically used for utility functions.

```
class MathUtil {
  static add(a, b) {
    return a + b;
  }

  static subtract(a, b) {
    return a - b;
  }
}

console.log(MathUtil.add(5, 3)); // Output: 8
console.log(MathUtil.subtract(5, 3)); // Output: 2
```

Key Concepts of OOP:

- Objects and Classes
- Encapsulation
- Inheritance
- Polymorphism

Prototype and Prototype Inheritance

JavaScript is a prototype-based language, which means that objects can inherit properties and methods from other objects. This inheritance is achieved through prototypes.

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}

Person.prototype.greet = function() {
  console.log('Hello, my name is ' + this.name);
};

const alice = new Person('Alice', 30);
alice.greet(); // Output: Hello, my name is Alice
```

Prototype Chain: The prototype chain is a series of links between objects. When trying to access a property or method, JavaScript will first look on the object itself. If it does not find it, it will look on the object's prototype, and so on, until it reaches the end of the chain (usually Object.prototype).

```
console.log(alice.hasOwnProperty('name')); // true
console.log(alice.hasOwnProperty('greet')); // false (inherited
from prototype)
console.log(alice.__proto__ === Person.prototype); // true
console.log(Person.prototype.__proto__ === Object.prototype); //
true
console.log(Object.prototype.__proto__); // null
```

Prototype Inheritance: Prototype inheritance allows an object to inherit properties and methods from another object.

```
function Animal(name) {
  this.name = name;
}

Animal.prototype.speak = function() {
  console.log(this.name + ' makes a sound.');
};

function Dog(name, breed) {
  Animal.call(this, name); // Call the parent constructor
  this.breed = breed;
}

Dog.prototype = Object.create(Animal.prototype); // Set up
inheritance
Dog.prototype.constructor = Dog; // Correct the constructor
pointer

Dog.prototype.speak = function() {
  console.log(this.name + ' barks.');
};

const rex = new Dog('Rex', 'Labrador');
rex.speak(); // Output: Rex barks.
```

ES6+ Features

ES6 (ECMAScript 2015) introduced several new features and enhancements to JavaScript, and subsequent versions (ES7, ES8, ES9, etc.) continued to add more. Here are some key ES6+ features in JavaScript:
Arrow Functions: Concise syntax for writing anonymous functions.

- **Classes:** Syntactical sugar over JavaScript's existing prototype-based inheritance.
- **Template Literals:** Improved syntax for string interpolation and multiline strings.
- **Destructuring:** Extracting values from arrays and objects into distinct variables.
- **Spread Syntax (...):** Expands iterable elements into multiple arguments or array elements.
- **Rest Parameters (...):** Collects multiple arguments into a single array parameter.
- **Default Parameters:** Assigns default values to function parameters.
- **Enhanced Object Literals:** Shorthand syntax for defining object properties and methods.
- **Promises:** Improved way to handle asynchronous operations, representing a value that might be available now, in the future, or never.
- **Async/Await:** Syntactic sugar over promises, making asynchronous code look synchronous.
- **Let and Const:** Block-scoped variable declarations, providing better scoping than var.
- **Map, Set, WeakMap, WeakSet:** New data structures for storing collections of data with unique or arbitrary keys/values.
- **Symbol:** Primitive data type representing unique values, useful for creating private object properties.
- **Iterators and Generators:** Provides a way to iterate over data structures such as arrays, strings, or custom objects.
- **Optional Chaining (?.) and Nullish Coalescing (??):** New operators for handling optional properties and default values more efficiently.
- **String Methods:** New methods added to the String prototype, such as startsWith(), endsWith(), includes(), and padStart().
- **Array Methods:** New array methods like find(), findIndex(), flat(), flatMap(), and fromEntries().

The 'this' keyword

The this keyword in JavaScript refers to the object it belongs to. Its value is determined by how a function is called, and it can vary depending on the context. Understanding how this works is crucial for writing effective and bug-free JavaScript code.

- **Global Context :** In the global execution context (outside of any function), this refers to the global object (window in browsers, global in Node.js).
- **Function Context :** In a regular function, this depends on how the function is called.
- **Method Context :** When a function is called as a method of an object, this refers to the object.
- **Constructor Context :** When a function is used as a constructor (with the new keyword), this refers to the newly created object.
- **Arrow Functions :** Arrow functions do not have their own this context; instead, this is lexically inherited from the enclosing scope at the time the arrow function is defined.

Common Pitfalls and How to Avoid Them

- **Losing this Context**
  - When passing a method as a callback or using it in an event handler, this might be lost.
  - Solution: Use bind, call, or apply to explicitly set this.
- **Arrow Functions in Methods**
  - Avoid using arrow functions as methods because they do not have their own this.
  - Solution: Use regular functions for methods.

calls(), apply(), bind()

call: This method is used to invoke a function with a specified this value and arguments provided individually.

```
function greet() {
  console.log(`Hello, ${this.name}!`);
}

const person = { name: 'John' };

greet.call(person); // Outputs: Hello, John!
```

• apply: Similar to call, but it accepts arguments as an array.

```
function greet(greeting) {
  console.log(`${greeting}, ${this.name}!`);
}
const person = { name: 'John' };
greet.apply(person, ['Hi']); // Outputs: Hi, John!
```

• bind: This method creates a new function that, when called, has its this keyword set to the provided value, with a given sequence of arguments.

```
function greet() {
  console.log(`Hello, ${this.name}!`);
}

const person = { name: 'John' };

const greetPerson = greet.bind(person);
greetPerson(); // Outputs: Hello, John!
```

Error Handling (try...catch, types of error)

Error handling in JavaScript is crucial for building robust applications. The try...catch statement is used to handle exceptions gracefully, ensuring that the program can continue running or fail in a controlled manner.

try...catch Statement

The try...catch statement allows you to define a block of code to test for errors while it is being executed. If an error occurs, the execution jumps to the catch block.

```
try {
  // Code that may throw an error
} catch (error) {
  // Code to handle the error
} finally {
  // Code to run regardless of an error (optional)
}
```

Types of Errors

JavaScript has several built-in error types to help identify what went wrong:

- **SyntaxError :** Occurs when the code contains a syntax mistake.
- **ReferenceError :** Occurs when trying to reference a variable that is not declared.
- **TypeError:** Occurs when a value is not of the expected type.
- **RangeError:** Occurs when a number is outside of an allowable range.
- **URIError:** Occurs when a global URI handling function is used incorrectly.
- **EvalError:** Occurs when the eval() function is used incorrectly. (Note: This error type is rarely used in modern JavaScript.)
- **AggregateError :** Used to represent multiple errors wrapped in a single error when multiple errors need to be reported.

Asynchronous Programming (Callback, Promise, Async-Await)

Asynchronous programming is a key feature in JavaScript, allowing tasks to run concurrently and improving the efficiency of applications, particularly for I/O-bound tasks such as network requests or file handling. JavaScript provides several ways to handle asynchronous operations: callbacks, promises, and async/await

1. Callbacks

A callback is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

```
function fetchData(callback) {
  setTimeout(() => {
    const data = { id: 1, name: 'Alice' };
    callback(data);
  }, 1000);
}

fetchData(data => {
  console.log('Data received:', data);
});
```

2. Promises

A promise is an object representing the eventual completion (or failure) of an asynchronous operation and its resulting value. Promises have three states: pending, fulfilled, and rejected.

```
const fetchData = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const data = { id: 1, name: 'Alice' };
      resolve(data);
    }, 1000);
  });
};

fetchData().then(data => {
  console.log('Data received:', data);
});
```

3. Async/Await

Async/await is syntactic sugar built on top of promises, making asynchronous code look and behave more like synchronous code.

```
const fetchData = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const data = { id: 1, name: 'Alice' };
      resolve(data);
    }, 1000);
  });
};

const handleData = async () => {
  const data = await fetchData();
  console.log('Data received:', data);
};

handleData();
```

Polyfills

A polyfill, short for "polymer fill," is a piece of code (usually JavaScript) that provides the functionality of newer APIs or features in older browsers that do not support them natively. Polyfills enable developers to use modern features of JavaScript and web standards while ensuring compatibility with older browsers.

Here's how a polyfill typically works:

- Feature Detection: Polyfills first check if the browser supports a particular feature by testing for the existence of related objects, methods, or properties.
- Conditional Loading: If the feature is not supported, the polyfill is loaded or executed to provide the missing functionality.

Fallback Implementation: The polyfill mimics the behavior of the modern feature using JavaScript code that replicates the functionality in older browsers

Regex

Regular expressions (regex or regexp) in JavaScript are patterns used to match character combinations in strings. They are highly useful for tasks such as searching, replacing, and validating text. Here's a concise overview of regex in JavaScript, including syntax, special characters, and methods.

Methods

- **test() :** Tests for a match in a string. Returns true or false.
- **match() :** Retrieves the matches in a string. Returns an array or null.
- **replace() :**Replaces matched substring(s) with a new substring.

split() : Splits a string into an array using the regex.

```
// test
let regex = /hello/;
console.log(regex.test('hello world')); // true

//match()
let regex = /\d+/;
console.log('There are 123 numbers'.match(regex)); // ["123"]

//replace()
let regex = /world/;
console.log('hello world'.replace(regex, 'there')); // "hello
there"

//split()
let regex = /\s+/;
console.log('hello world'.split(regex)); // ["hello", "world"]

//Example with Quantifiers
let regex = /\d{2,4}/;
console.log('12345'.match(regex)); // ["1234"]
```

Flags

- **g:** Global match (find all matches rather than stopping after the first match)
- **i:** Case-insensitive matching
- **m:** Multi-line matching

Special Characters

- **.:** Matches any character except newline
- **\d:** Matches a digit (0-9)
- **\w:** Matches a word character (alphanumeric + underscore)
- **\s:** Matches a whitespace character (space, tab, newline)
- **^:** Matches the beginning of the string
- **$:** Matches the end of the string
- **\:** Escapes a special character

Character Classes

- **[abc]:** Matches any one of the characters a, b, or c
- **[^abc]:** Matches any character not in the brackets
- **[a-z]:** Matches any character in the range a to z

Quantifiers

- ***:** Matches 0 or more times
- **+:** Matches 1 or more times
- **?:** Matches 0 or 1 time
- **{n}:** Matches exactly n times
- **{n,}:** Matches at least n times
- **{n,m}:** Matches between n and m times