# JS EMG

## Table Of Contents



## Basics of JS

- Basic JavaScript Concepts
- Variables
- Data Types
- Operators
- Functions
- Closures
- Arrow Functions
- Rest Arguments (ES6)
- Conditional Statements
- Loops
- Objects
- Spread and Rest Operators
- Arrays
- Ternary Operators
- Optional Chaining
- Array Methods (map, filter, reduce)
- Function Calls
- Function Parameters and Arguments
- Debugging Techniques
- Template Literals
- Optional Chaining with Nullish Coalescing

```
// 1.js
// To run js file > open terminal > node file_name.js

// Single Line Comment

//JavaScript is your browser has a built-in execution environment.

/*
Multi
-
Line
Comment
```

```
*/

/*
1. Basic JavaScript Concepts:
   - Variables and Data Types
   - Operators
   - Control Flow (if statements, loops)
   - Functions
   - Objects and Arrays
   - DOM Manipulation (in 1a.html (which contains html with script ;0 ))
   - Ternary operators
*/

// variable
// Declare a variable
let message = "Hello World!";
let name = "Siddhant Bali Computer Science Sorcerer "

// Datatypes
let num = 1010; // Number
let text = "Some text"; // String
let isTrue = true; // Boolean
let data; // Undefined
let myArray = [1, 2, 3]; // Array
let myObject = { key: "value" }; // Object (dict. in python)
let persona = { name: "Bob", age: 30 }; // Object

// Print the variable
console.log(message);

/*

;

In JavaScript, the semicolon (;) is used to terminate statements.
While JavaScript does have automatic semicolon insertion (ASI),
it's a good practice to include semicolons explicitly to avoid potential
issues,
especially in larger codebases.

It's important to note that in modern JavaScript,
there are contexts (such as at the end of a block)
where a semicolon is not strictly required. However,
including them is a recommended practice to avoid potential issues.

*/

// Operators
let x = 10;
let y = 5;
let sum = x + y; // Addition
let difference = x - y; // Subtraction
let product = x * y; // Multiplication
let quotient = x / y; // Division
```

```
let isEqual = x === y; // Strict equality === ,not ==
let isNotEqual = x !== y; // Strict inequality
let isTrue1 = true;
let isFalse = false;
let andResult = isTrue && isFalse; // Logical AND
let orResult = isTrue || isFalse; // Logical OR
let notResult = !isTrue; // Logical NOT

// Printing results
console.log("Sum:", sum);  *have default new line character "\n"
console.log("Difference:", difference);
console.log("Product:", product);
console.log("Quotient:", quotient);
console.log("Is Equal:", isEqual);
console.log("Is Not Equal:", isNotEqual);
console.log("Logical AND Result:", andResult);
console.log("Logical OR Result:", orResult);
console.log("Logical NOT Result:", notResult);


// Function declaration
function greet(usrname){
    console.log("Hello "+usrname+" in this tutorial program ");
    console.log(";)");
}

function test(){
    console.log("test done ;)");
}


// Function call
greet(name);

*Consise
*Arrow func. are too anonymous
*Callback functions, meaning they are passed as an argument to another function
*Assign to a Variable

//Anonymous Function
var greet2 = function(usrname){return "Hello "+usrname+" in this tutorial
program ";};

//Immediately Invoked Function Expression (IIFE)
const result1 = (function (usrname) {return "Hello "+usrname+" in this
tutorial program ";})("Siddhant Bali");
console.log(result1);

//A powerful feature of JavaScript is you can actually create functions
within other functions and even return them!
function createFunction() {
  return f;
  function f(a, b) {
      const sum = a + b;
      return sum;
  }
}
const f = createFunction();
console.log(f(3, 4)); // 7
```

```
// Closures : A closure is a function that has access to its outer function
scope even after the outer function has returned. This means a closure can
remember and access variables and arguments of its outer function even
after the function has finished.
function createAdder(a) {
  function f1(b) {
      const sum = a + b;
      return sum;
  }
  return f1;
}
const f1 = createAdder(3);   // stores func with a=3
console.log(f(4)); // 7


// Arrow Functions (ES6)  //ECMAScript ,is a major update to the JavaScript programming language.
let Multiplication = (a,b) => a*b;

// Arrow Function's Omit Return {f2 and f3 are same}
const f2 = (a, b) => {
  const sum = a + b;
  return sum;
};
console.log(f2(3, 4)); // 7

const f3 = (a, b) => a + b; // way shorterrrr !!!
console.log(f3(3, 4)); // 7

/*
Functions vs arrow functions

There are 3 major differences between arrow syntax and function syntax.

More minimalistic syntax.
This is especially true for anonymous functions and single-line functions.
For this reason, this way is generally preferred when passing short
anonymous functions to other functions.

No automatic hoisting.
You are only allowed to use the function after it was declared. This is
generally considered a good thing for readability.

Can't be bound to this, super, and arguments or be used as a constructor.
These are all complex topics in themselves but the basic takeaway should be
that arrow functions are simpler in their feature set. You can read more
about these differences here.

*/

// Rest Arguments ... (ES6) (means keep everything in an array)
function f11(...args) {
  const sum = args[0] + args[1];
  return sum;
```

```
}
console.log(f11(3, 4)); // 7
// wrapper function

function log(inputFunction) {  //takes input of func
  return function(...args) { //return another func
      console.log("Input", args);
      const result = inputFunction(...args); //now the input function is played
      console.log("Output", result);
      return result;
  }
}
const f111 = log((a, b) => a + b);
f(1, 2); // Logs: Input [1, 2] Output 3
f111(1, 2)
// Conditional Statements
// if,else,else if
if (num > 0) {
  console.log("Positive number");
} else if (num < 0) {
  console.log("Negative number");
} else {
  console.log("Zero");
}

// for loop
for (let index = 0; index < myArray.length; index++) {
    const element = myArray[index];
    console.log(element)
}

// While loop
let counter = 0;
while (counter<=5) {
    console.log(counter);
    counter++;

}

// Object
let person = {
    firstName: "John",
    lastName: "Doe",
    age: 30,
    fullName: function () {
      return this.firstName + " " + this.lastName;
    },
  };

const age_person = person.age;
const {a1,a2,a3} = person ; //deconstruct (easy way to assign stuff)

let car= {model: "Swift Dezire",company:"Maruti Suzuki",desc:function()
{return this.model+" by "+this.company;}}; // another example
```

```
const b1 = "hi";
const b2 = "hola";
const b = {
  b1, //it means b1 = b1 ;
  b2, //b2 = b2 ;

};

// ... Spread Operator (means keep everything form
datastructures(obj,arr,etc))
let person2 = {...person,firstName:"Daisy"};
// above means that person2 will have exact all stuff of person and
firstName is Daisy
const name2=[1,2,3];
const name3=[...name2,4]; //it means name3 will have exact all stuff of
name2 and also an entry of 4




// Accessing object properties and calling a method
console.log(person.firstName); // John
console.log(person.fullName()); // John Doe

console.log(car);
/*
{
  model: 'Swift Dezire',
  company: 'Maruti Suzuki',
  desc: [Function: desc]
}


*/
console.log(car.model);
console.log(car.company);
console.log(car.desc());


// Array
let fruits = ["apple", "banana", "orange"];

// Iterating through an array
for (let i = 0; i < fruits.length; i++) {
  console.log(fruits[i]);
}

// Array methods
fruits.push("grape"); // Add an element to the end
fruits.pop(); // Remove the last element
fruits.unshift("kiwi"); // Add to the beginning
```

```
// Iterating through an array
for (let i = 0; i < fruits.length; i++) {
  console.log(fruits[i]);
}

// Array methods
fruits.push("grape"); // Add an element to the end
fruits.pop(); // Remove the last element
fruits.unshift("kiwi"); // Add to the beginning
fruits.shift(); // Remove from the beginning
console.log(fruits); // ["apple", "banana", "orange"]

// Using forEach to iterate through each element in the 'fruits' array and
print it.
fruits.forEach(function (fruit) {
  console.log("Current Fruit:", fruit);
});

// Using filter to create a new array ('filteredFruits') by excluding the
"orange" fruit.
let filteredFruits = fruits.filter(function (fruit) {
  // The filter callback returns true for elements that should be included
in the filtered array.
  // In this case, it excludes "orange" from the new array.
  return fruit !== "orange";
});

// Displaying the filtered array of fruits after excluding "orange".
console.log("Filtered Fruits (excluding 'orange'):", filteredFruits);

// Ternary operators
// condition ? expression_if_true : expression_if_false;
let age = 20;
let message2 = (age >= 18) ? 'You are an adult' : 'You are a minor';

console.log(message2);
// Output: 'You are an adult' (since age is 20, which is greater than or
equal to 18)

let Qual = 1;
let ProdQual = (Qual=1)?'Premium': (Qual=2)?'Medium':'Worst';
console.log(ProdQual);

let num1 = 10;

// Check if num1 is greater than 0      // nested
let result = (num1 > 0)
  ? 'Positive' // If true, set result to 'Positive'
  : (num1 < 0)
    ? 'Negative' // If false, check if num1 is less than 0. If true, set
result to 'Negative'
    : 'Zero'; // If false, set result to 'Zero'

console.log(result);
```

```
// Output: 'Positive' (since num1 is 10, which is greater than 0)

const Component = () => {return age>10?<div>Bhati</div>:<div>Bali</div>;};
 // application of ternary

let names = ["Bali","Bhaskar","Bhati","Bali","Bali"];
// i want to make each element have 1 at last Bali1
// normal way,for loop
// mentos way

names.map((name)=>{
    return name + "1"
});
       // its application
names.map((name)=>{
    return <h1>{name}</h1>
});

names.filter((name)=>{
    return name !== "Bali"
});

// .map()
// .filter()
// .reduce()


const name1 = data?.person?.name;
/*
"?"
: This is the optional
chaining operator. It allows you to access
properties of an object without the need
to explicitly check if each level of the
object hierarchy is defined
(i.e., not null or undefined).
If any intermediate property in the chain
is null or undefined, the entire expression
evaluates to undefined instead of throwing an
error.


*/
```

- Accessing Elements
    - `getElementById()`
    - `getElementsByClassName()`
    - `getElementsByTagName()`
    - `querySelector()`
    - `querySelectorAll()`

- Modifying Content
    - `innerHTML`
- Changing Styles
    - `style` property
    - `classList.add()`
- Creating and Deleting Elements
    - `createElement()`
    - `appendChild()`
- Event Handling
    - `addEventListener()`
- Traversing the DOM
    - `firstChild`
    - `lastChild`
    - `nextSibling`
    - `previousSibling`
- Debugging
    - `console.log()`
- Multi-line Comments in JavaScript
    - `/* Comment */`
- Single-line Comments in HTML
    - `<!-- Comment -->`

```
<!-- 1a.html -->

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>DOM Manipulation Example</title>
    <style>
        .highlight {
            color: red;
            font-weight: bold;
        }
    </style>
</head>
<body>
```

```
    <!-- Accessing Elements -->
    <h2 id="myId">Accessing Elements</h2>
    <div class="myClass">Element by Class</div>
    <div>Element by Tag</div>
    <div class="myClass">First Element by Selector</div>

    <!-- Modifying Content -->
    <h2>Modifying Content</h2>
    <div id="contentElement">Original Content</div>

    <!-- Changing Styles -->
    <h2>Changing Styles</h2>
    <div id="styleElement">Style Example</div>

    <!-- Creating and Deleting Elements -->
    <h2>Creating and Deleting Elements</h2>
    <div id="parentElement">Parent Element</div>

    <!-- Event Handling -->
    <h2>Event Handling</h2>
    <button id="clickButton">Click me</button>

    <!-- Traversing the DOM -->
    <h2>Traversing the DOM</h2>
    <div id="traverseParent">
        <div>Child 1</div>
        <div>Child 2</div>
        <div>Child 3</div>
    </div>

    <!-- Debugging -->
    <h2>Debugging</h2>

 <!-- Html dont have multine comment ;( -->
 <!-- Using JS in Html ;)  -->
    <script>
        /*
        Html dont have multine comment ;(

        xdg-open index.html //gnome
        kde-open index.html //kde
        firefox index.html  # Replace "firefox" with the name of your
preferred browser



        */
        /  DOM
         * DOM (Document Object Model) manipulation is a fundamental
concept in web development
         * using JavaScript. It allows you to interact with the HTML and
XML documents in a web page,
         *  dynamically updating content, structure, and style. Here's an
```

```
in-depth explanation for beginners

        - The DOM is a hierarchical tree-like structure representing the
structure of a document.
        - HTML elements are nodes in this tree, and each node can have
child nodes.
        - The topmost node is the document node, representing the entire
document.

         *
         *
         */




        // Accessing Elements
        const myElement = document.getElementById('myId');
        const elementsByClass = document.getElementsByClassName('myClass');
        const elementsByTag = document.getElementsByTagName('div');
        const firstElement = document.querySelector('.myClass');
        const allElements = document.querySelectorAll('.myClass');

        // Modifying Content
        const contentElement = document.getElementById('contentElement');
        contentElement.innerHTML = '<p>New content</p>';

        // Changing Styles
        const styleElement = document.getElementById('styleElement');
        styleElement.style.color = 'blue'; // change colour to blue
        styleElement.classList.add('highlight'); // to bold i guess

                                          + Appending ;)
        // Creating and Deleting Elements
        const parentElement = document.getElementById('parentElement');
        const newElement = document.createElement('p');
        newElement.innerText = 'New Element';
        parentElement.appendChild(newElement);

        // Event Handling
        const clickButton = document.getElementById('clickButton');
        clickButton.addEventListener('click', function () {
            clickButton.innerText = 'Button Clicked!';
        });

        // here if we use console ,so it will run at browser's console : go
to inspect Ctrl+Shift+I

        // Traversing the DOM
        const traverseParent = document.getElementById('traverseParent');
        console.log(traverseParent);
        const firstChild = traverseParent.firstChild;
        console.log(firstChild);
        const lastChild = traverseParent.lastChild;
```

// Get the element you want to remove
const elementToRemove =
document.getElementById('parentElement');
// Remove the element from the DOM
elementToRemove.remove();

```
        console.log(lastChild);
        const secondChild = firstChild.nextSibling;
        console.log(secondChild);
        const previousChild = lastChild.previousSibling;
        console.log(previousChild);

        // Debugging
        console.log('Debugging information');

    </script>
</body>
</html>
```

```
// DOM
//       [document]
//          |
//       [<html>]
//          |
//   +--------+--------+
//   |                 |
// [<head>]         [<body>]
//   |                 |
// [<title>]      +----+----+
//   |            |         |
// [text]       [<h1>]     [<p>]
//                |         |
//              [text]    [text]

// DOM, Parent, Child, and Sibling Explained
// The 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. The DOM represents the document as nodes and objects.

// In the example above, document is the root of the tree. Let's break down the relationships:

// Parent: A node that has other nodes branching off of it. In the diagram, <html> is the parent of both <head> and <body>.

// Child: A node that is directly connected to another node, which is its parent. <head> and <body> are both children of <html>.

// Sibling: Nodes that share the same parent. <head> and <body> are siblings because they both have <html> as their parent.
```

## ES6 and Modern JS

- ES6 and Modern JavaScript:
  - Arrow Functions
  - Destructuring Assignment
  - Template Literals
  - Let and Const
  - Classes and Inheritance
  - Modules (import/export)
  - Promises
  - Default Parameters
  - Rest and Spread Operators
  - Enhanced Object Literals
  - Symbol
- Spread Syntax
  - Expanding elements of an array or properties of an object.
- Rest Syntax
  - Collecting remaining elements or properties into a single variable.
- Traditional Function
  - Standard function declaration.
- Anonymous Function
  - Functions without names, often used for short-lived operations.
- Arrow Function
  - Concise function syntax with no `this` binding.
- Array Destructuring
  - Extracting values from arrays into variables.
- Object Destructuring
  - Extracting properties from objects into variables.
- Template Literals
  - String literals with embedded expressions and multiline support.
- `var`, `let`, and `const`
  - `var`: Function-scoped, hoisted, can be reassigned.
  - `let`: Block-scoped, hoisted but not initialized, can be reassigned.
  - `const`: Block-scoped, hoisted but not initialized, cannot be reassigned.

- Class and Inheritance
  - Classes to create and extend objects, simplifying object-oriented code.
- Static Methods
  - Methods that belong to the class itself, not instances of the class.
- Getters and Setters
  - Special methods for getting and setting object properties.
- Backtick vs Single Quote
  - Backticks are for template literals, single quotes for regular strings.
- Creating a Class
  - Basic structure of defining classes and methods in JavaScript.
- Extending a Class
  - Inheritance in JavaScript using the `extends` keyword.

```
// 2.js
/*
2. ES6 and Modern JavaScript:
   - Arrow functions
   - Destructuring assignment
   - Template literals
   - Let and const
   - Classes and inheritance
   - Modules (import/export)

   ES6, short for ECMAScript 2015, is the sixth edition of the ECMAScript
standard,
   which is the scripting language specification that JavaScript is based
on.
   ES6 introduced significant enhancements and new features to the
JavaScript language,
   providing developers with more tools and syntactic sugar to write clean,
efficient, and expressive code.

When people refer to "modern JavaScript," they are generally talking about
the practices
and features introduced in and after ES6. Modern JavaScript encompasses the
latest ECMAScript specifications,
as well as contemporary coding patterns and best practices.

Key features introduced in ES6 and considered part of modern JavaScript
include:

1. Arrow Functions: A concise syntax for writing functions, especially
useful for
short anonymous functions and functions with a simple body.

2. Destructuring Assignment: A syntax that allows unpacking values from
arrays or
properties from objects into distinct variables, making code more concise.

3. Template Literals: A more expressive way to concatenate strings and
include expressions
```

```
inside strings using backticks (`).

4. let and const: `let` and `const` introduced block-scoping, replacing the
sometimes
confusing behavior of `var`. `let` allows variable reassignment, while
`const` declares constants.

5. Classes and Inheritance: The introduction of class syntax, making it
easier to
create and extend object-oriented code, simplifying the prototype chain.

6. Modules (import/export): A standardized way to organize code into
reusable and
maintainable modules, promoting a more modular approach to building
applications.

7. Promises: A built-in mechanism for handling asynchronous operations,
providing a more structured and readable way to deal with asynchronous code
than callback functions.

8. Default Parameters: Allows function parameters to have default values,
reducing the need for boilerplate code.

9. Rest and Spread Operators: The `...` syntax for both rest and spread
operations,
providing a concise way to handle variable arguments in functions or spread
elements in arrays.

10. Enhanced Object Literals: Shorthand syntax for defining methods and
properties in object literals,
making object creation more concise.

11. Symbol: A new primitive data type, providing a way to create unique
identifiers,
which is useful for creating private object properties and avoiding naming
collisions.

These features, along with subsequent additions in newer ECMAScript
specifications
(ES7, ES8, ES9, and so on), collectively define what is considered modern
JavaScript.
Developers often use these features to write more efficient, readable, and
maintainable code.
It's essential for JavaScript developers to stay up-to-date with modern
practices to leverage
the full capabilities of the language.
   */

// Spread Syntax Example
// It is used to expand elements of an iterable (like an array) or
properties of an object.
const array1 = [1, 2, 3];
const array2 = [...array1, 4, 5, 6]; // Spread elements of array1 into
array2
```

```
console.log(array2); // Output: [1, 2, 3, 4, 5, 6]

const obj1 = { key1: 'value1', key2: 'value2' };
const obj2 = { ...obj1, key3: 'value3' }; // Spread properties of obj1 into
obj2
console.log(obj2); // Output: { key1: 'value1', key2: 'value2', key3:
'value3' }

// Rest Syntax Example
// It is used to collect the remaining elements of an array or properties
of an object into a single variable.
const [first1, second1, ...rest] = [1, 2, 3, 4, 5];
console.log(first1); // Output: 1
console.log(second1); // Output: 2
console.log(rest); // Output: [3, 4, 5]

const { key1, ...restObj } = { key1: 'value1', key2: 'value2', key3:
'value3' };
console.log(key1); // Output: 'value1'
console.log(restObj); // Output: { key2: 'value2', key3: 'value3' }


// Traditional function
function add(a, b) {
   return a + b;
}


function Multi(a,b){return a*b;}

// Anonymous Function
<button
  onClick={() => {
    console.log("Hi");
  }}
>
  Click me
</button>;

// Arrow function
const addArrow = (a, b) => a + b;
const MultiArrow = (a,b) => a*b;

// Case 1: Single parameter
const square = x => x * x;

// Case 2: No parameters
const greet = () => console.log('Hello!');

// Case 3: Returning an object (maybe more than one)
const createPerson = (name, age) => ({ name, age });

// Case 4: Using with array functions
const numbers = [1, 2, 3];
```

```
const squaredNumbers = numbers.map(x => x * x);
/*
 The map function is used to iterate over each element
 of the numbers array and apply a function to each element,
 creating a new array based on the results.
*/

// one more Eg:
const MyComponent = () => {return <div></div>;};

// Array destructuring
const numbers1 = [1, 2, 3];
const [first, second, third] = numbers1;
const [a,b,c] = numbers1;

// Object destructuring
const person = { name: 'John', age: 30 };
const { name, age } = person;

// Case 1: Swapping variables
let a1 = 1, b1 = 2;
[a1, b1] = [b1, a1];

// Case 2: Nested destructuring
const user = {
    name: 'Alice',
    info: {
        age: 25,
        country: 'Wonderland'
    }
};
const { name1, info: { age1, country } } = user;

// Case 3: Default values
const settings = { color: 'blue' };
const { theme = 'default', color } = settings;

// Template Literals
const name3 = 'World';
const greeting = `Hello, ${name3}!`;
const bye = `Bye Bye, ${name3}`;

// Case 1: Multiline strings
const multilineString = `
    This is a
    multiline string.
`;

// Case 2: Expressions inside templates
const a2 = 5, b2 = 10;
const result = `The sum of ${a2} and ${b2} is ${a2 + b2}.`;

// Case 3: Tagged templates
function myTag(strings, ...values) {
```

```
    // Your custom logic here
    return `${strings[0]}${values.join('')}${strings[1]}`;
}

const taggedResult = myTag`Hello, ${name1}!`;

function greetf(name4, age2) {
    return `Hello ${name4} in this ${age2} year`;
}

console.log(greetf('Siddhant Bali', 2023));


// Using var
var x = 10;
var x1 =69;

// Using let (block-scoped)
// let is just var but block-scoped version ;0
// block-scoped means if they are under { } so ,their existence will be
under that only
let y = 20;

// Using const (block-scoped, and cannot be reassigned)
const z = 30;

// Case 1: Reassigning with let
let counter = 0;
counter = 1;

// Case 2: Const with objects and arrays
//(reference cannot be changed, but contents can)
const numbers2 = [1, 2, 3];
numbers2.push(4);

// Case 3: Temporal dead zone
//console.log(a3); // ReferenceError: Cannot access 'a' before
initialization
let a3 = 5;

/*
Certainly! Here's a summary of the information about `var`, `let`, and
`const`:

- `var`:
  - Scope: Function-scoped or globally scoped.
  - Hoisting: Hoisted to the top of the scope, allowing use before
declaration.
  - Reassignment: Can be reassigned and updated.

- `let`:
  - Scope: Block-scoped (limited to the block where it is defined).
  - Hoisting: Hoisted to the top of the block but not initialized,
  so cannot be used before declaration.
```

```
  - Reassignment: Can be reassigned and updated.

- `const`:
  - Scope: Block-scoped.
  - Hoisting: Hoisted to the top of the block but not initialized.
  - Reassignment: Cannot be reassigned after the initial assignment,
  making it a constant value.

Recommendations:
- Use `var` for older code or if function-scoping is specifically required.
- Prefer `let` for variables that may need to be reassigned within a block.
- Use `const` for constants and values that should not be reassigned after
the initial assignment, providing a more robust and predictable code
structure.

*/

// Creating a class
class Animal {
    constructor(name) {
        this.name = name;
    }

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

    run(){console.log(`${this.name} run like a ${this.name}`);}

}
// ` vs '
/*
In JavaScript, the backtick (`) and single quote (') are both used to
define strings,
but they have different purposes.
Single quotes are traditionally used to define strings in JavaScript.
However, if you want to include a single quote within a string defined with
single quotes,
you need to escape it with a backslash:

The backtick, introduced in ECMAScript 6 (ES6), is used for template
literals.
Template literals allow for embedded expressions and multiline strings:
You can also include variables and expressions directly within a template
literal:
This is useful for creating more complex strings in a concise and readable
way.*/

// Extending a class
class Dog extends Animal {
    speak() {
        console.log(`${this.name} barks.`);
    }
}
```

```
const myDog = new Dog('Buddy');
myDog.speak();

// Case 1: Static methods
class MathOperations {
    static add(x, y) {
        return x + y;
    }
}

const sum1 = MathOperations.add(3, 4);

// Case 2: Getters and setters
class Circle {
    constructor(radius) {
        this.radius = radius;
    }

    get diameter() {
        return this.radius * 2;
    }

    set diameter(value) {
        this.radius = value / 2;
    }
}

const myCircle = new Circle(5);
console.log(myCircle.diameter); // Getter
myCircle.diameter = 12; // Setter
```

- mathFunctions.js
  - Named Exports:
    - `export const add = (a, b) => a + b;`
    - `export const subtract = (a, b) => a - b;`
    - `export const multiply = (a, b) => a * b;`
    - `export const mod = (a,b) => a % b;`
  - Default Export:
    - `const divide = (a, b) => a / b;`
    - `export default divide;`
  - Purpose: Used to split code into separate modules for better reusability and maintainability.

```
// mathFunction.js
// Named exports
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;
export const multiply = (a, b) => a * b;
export const mod = (a,b)=> a%b ;
```

```
// Default export
// Default export does not need curly braces
const divide = (a, b) => a / b;
export default divide;

/*
the import/export syntax is used to create modular
code by allowing you to split your code into separate files
and reuse them as needed.
*/
```

- app.js
    - Import Named Exports:
        - `import { add, subtract, multiply, mod } from './mathFunctions';`
    - Import Default Export:
        - `import divide from './mathFunctions';`
    - Alias Import:
        - `import { add as sum } from './mathFunctions';`
    - Import Everything:
        - `import * as math from './mathFunctions';`
    - Re-export:
        - `export { add, subtract } from './mathFunctions';`
    - Dynamic Import:
        - `import('./mathFunctions').then(mathModule => { ... });`
    - Using Imported Functions:
        - `console.log(add(2, 3));` // Output: 5
        - `console.log(subtract(5, 2));` // Output: 3
        - `console.log(multiply(2, 4));` // Output: 8
        - `console.log(divide(10, 2));` // Output: 5
        - `console.log(sum(2, 3));` // Output: 5
        - `console.log(math.multiply(3, 3));` // Output: 9
        - `console.log(mod(5, 2));` // Output: 1
- CommonJS Import Example
    - `const { add } = require('./mathFunctions');`
    - Purpose: Used for importing in CommonJS-based environments (e.g., Node.js).

```
// app.js
// Import named exports and default export
import { add, subtract, multiply ,mod } from './mathFunctions';
import divide from './mathFunctions'; // Default export does not need curly
braces

// Alias import
import { add as sum } from './mathFunctions';
```

```
// Import everything
import * as math from './mathFunctions';

// Re-export from another module
export { add, subtract } from './mathFunctions';

// Using the imported functionalities
console.log(add(2, 3));         // Output: 5
console.log(subtract(5, 2));    // Output: 3
console.log(multiply(2, 4));    // Output: 8
console.log(divide(10, 2));     // Output: 5
console.log(sum(2, 3));          // Output: 5
console.log(math.multiply(3, 3)); // Output: 9
console.log(mod(5,2));

// Dynamic import
import('./mathFunctions').then(mathModule => {
  console.log(mathModule.add(4, 5)); // Output: 9
});

// Re-exported functionalities
console.log(add(7, 2));       // Output: 9
console.log(subtract(10, 5)); // Output: 5

// CommonJS
const { add } = require('./mathFunctions');
```

- index.html
  - Module Script:
    - `<script type="module" src="app.js"></script>`
    - This allows `app.js` to use modern JavaScript features like `import` and `export`.
  - Purpose: The HTML file to load the JavaScript module into the browser.

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Module Example</title>
</head>
<body>
  <script type="module" src="app.js"></script>
</body>
</html>
```

## Asynchronous JavaScript

- Asynchronous JavaScript Concepts:
    - Callbacks
    - Promises
    - Async/Await
- Callbacks
    - A function passed as an argument to be executed after another function finishes.
    - Example: `setTimeout`, `fs.readFile`
- Promises
    - An object representing the eventual completion (or failure) of an asynchronous operation.
    - States of a Promise: Pending, Resolved, Rejected
    - `.then()` for success, `.catch()` for failure
- Async/Await
    - Syntactic sugar for working with promises, making asynchronous code look synchronous.
    - `async` makes a function return a promise, and `await` pauses the execution until the promise is resolved.
- Common Use Cases
    - Network Requests: e.g., Fetching data from a server.
    - File Operations: e.g., Reading/writing files without blocking the main thread.
    - Timers: e.g., Using `setTimeout` or `setInterval` to execute code after a delay.

```
// 3.js
// 3. **Asynchronous JavaScript:**
//    - Callbacks
//    - Promises
//    - Async/Await

/*
an asynchronous operation is a task that doesn't block the entire
program while it's being executed. Instead of waiting for the operation
to finish, the program continues with other tasks and handles the
result of the asynchronous operation later. Examples include network r
equests, file operations, and timed events. Asynchronous programming
in JavaScript uses techniques like callbacks, promises, and async/await
to manage and organize this type of code.

Imagine you are at a coffee shop.
If you place an order for a coffee and the
barista immediately hands you the coffee, that's a
synchronous operation. The program (or your coffee order)
is blocked until the operation (or the coffee preparation) is complete.

On the other hand, if you place an order and receive a buzzer,
allowing you to sit down while waiting for your coffee, that's an
```

```
asynchronous operation. You can do other things (like reading a book
    or checking your phone) while the coffee is being prepared. Once
    your coffee is ready, the buzzer alerts you, and you can pick up
    your coffee. During this time, you weren't blocked and could
    perform other tasks.

In JavaScript, common examples of asynchronous operations include:

Network Requests: When your program makes a
request to a server for data, it often takes some time to get a response.
Meanwhile, your program can continue doing other things.

File Operations: Reading or writing to a file can take time,
especially in a large file. Asynchronous file operations allow
your program to continue its execution while waiting for the file
operation to complete.

Timers: Functions like setTimeout or setInterval allow you to
schedule code to run after a specified amount of time, without
blocking the rest of your program.

Understanding and effectively working with asynchronous
operations is crucial in web development, where interactions
with servers, user interfaces, and external resources often
involve some form of delay. Concepts like callbacks, promises,
and async/await are tools in JavaScript to manage and handle
asynchronous code in a more readable and maintainable way.
```

async: Marks a function as asynchronous and makes it return a Promise.

await: Pauses the function until a Promise resolves, then continues with the result.

```
*/
```

Callbacks: Functions passed as arguments to handle asynchronous operations, but can lead to messy nesting (callback hell).
Promises: A cleaner alternative to callbacks, allowing chaining of .then() and .catch() for success and error handling.

```
// Callbacks
/*
Callbacks:
A callback is a function that is passed as an
argument to another function and is executed after
the completion of some asynchronous operation.
*/

// Example callback function
function myCallback(result) {
    console.log("Callback executed with result: " + result);
  }

  // Function with a callback
  function performAsyncOperation(data, callback) {
    // Simulating an asynchronous operation
    setTimeout(function () {
      const result = data * 2;
      callback(result);
    }, 1000); // 1000 milliseconds delay
  }

  // Using the callback
```

This code demonstrates how callbacks work in JavaScript for handling asynchronous operations.

myCallback(result) is a simple function that logs the result passed to it.

performAsyncOperation(data, callback) is a function that simulates an asynchronous operation using setTimeout(). After 1 second, it computes data * 2 and calls the provided callback function with the result.

performAsyncOperation(5, myCallback) calls the function with 5 as input. After the 1-second delay, myCallback is executed, logging the result of 5 * 2, which is 10.

In summary, a callback is a function passed as an argument to another function, and it gets executed once the asynchronous operation is complete. This is how JavaScript traditionally handles async tasks, but it can lead to callback hell if multiple nested callbacks are used.

```
  performAsyncOperation(5, myCallback);

// Promises
/*
Promises provide a cleaner way to handle asynchronous operations.
A promise represents the eventual completion or failure of an
asynchronous operation.
*/
// Example using promises
function performAsyncOperation1(data) {
    return new Promise(function (resolve, reject) {
      // Simulating an asynchronous operation
      setTimeout(function () {
        const result = data * 2;
        resolve(result);
        // or reject("Operation failed!"); for error
      }, 1000); // 1000 milliseconds delay
    });
  }

  // Using the promise
  performAsyncOperation1(5)
    .then(function (result) {
      console.log("Promise resolved with result: " + result);
    })
    .catch(function (error) {
      console.error("Promise rejected with error: " + error);
    });

// Example using promises with reject case
function performAsyncOperation11(data) {
    return new Promise(function (resolve, reject) {
      // Simulating an asynchronous operation
      setTimeout(function () {
        if (data > 0) {
          const result = data * 2;
          resolve(result);
        } else {
          reject("Operation failed! Input should be greater than 0.");
        }
      }, 1000); // 1000 milliseconds delay
    });
  }

  // Using the promise
  performAsyncOperation11(5)
    .then(function (result) {
      console.log("Promise resolved with result: " + result);
    })
    .catch(function (error) {
      console.error("Promise rejected with error: " + error);
    });

  // Example with rejection
```

This code demonstrates how Promises are used to handle asynchronous operations in a cleaner and more manageable way compared to callbacks.

Key Concepts:
Promises represent the eventual completion (or failure) of an asynchronous operation. A Promise has three states:

Pending — Initial state, operation is not finished.

Resolved (Fulfilled) — Operation completed successfully.

Rejected — Operation failed (error occurred).

Breakdown:
performAsyncOperation1(data):
This function returns a Promise.

Inside the Promise, it simulates an asynchronous operation using setTimeout() to wait 1 second.

If the operation is successful, it calls resolve (result) to pass the result.

If there's an error, reject(error) can be used (though it's not used in this case).

Using Promises:

.then() is used to handle a successful resolution of the Promise, where the result is logged to the console.

.catch() is used to handle a rejection (error), and in this case, logs an error message if the operation fails.

performAsyncOperation11(data):

This function also returns a Promise, but with added validation:
If data > 0, it resolves with data * 2.

If data <= 0, it rejects the Promise with an error message.

Using the Promise with rejection:

When the function is called with 5, the Promise resolves successfully.
When the function is called with -2, the Promise rejects and logs an error message.

Summary:
Promises provide a more structured and readable way to handle asynchronous operations compared to callbacks.

```
  performAsyncOperation11(-2)
    .then(function (result) {
      console.log("Promise resolved with result: " + result);
    })
    .catch(function (error) {
      console.error("Promise rejected with error: " + error);
    });


// Async/Await
/*
Async/await is a syntax sugar built on top of promises,
 making asynchronous code look more like synchronous code.

*/
// Example using async/await
async function performAsyncOperation2(data) {
    return new Promise(function (resolve) {
      // Simulating an asynchronous operation
      setTimeout(function () {
        const result = data * 2;
        resolve(result);
      }, 1000); // 1000 milliseconds delay
    });
  }

  // Using async/await
  async function fetchData() {
    try {
      const result = await performAsyncOperation2(5);
      console.log("Async/Await result: " + result);
    } catch (error) {
      console.error("Async/Await error: " + error);
    }
  }

  // Calling the async function
  fetchData();
```

This code demonstrates async/await for handling asynchronous operations in JavaScript.

performAsyncOperation2(data) is a function that returns a Promise, simulating an asynchronous operation (multiplying a number after a 1-second delay).

fetchData() is an async function, which means it can use await to pause execution until the performAsyncOperation2() Promise resolves. This makes it appear as if the asynchronous operation is happening synchronously (like a normal function).

Inside fetchData(), await pauses the function until performAsyncOperation2(5) completes. Once it resolves, the result is printed to the console.

In short, async/await simplifies working with asynchronous code by making it more readable and less prone to complex chaining or callback issues.