JavaScript Programming Language

Mensi Mohamed Amine

Abstract

JavaScript is a dynamic, high-level programming language primarily used for creating interactive web applications. Its versatility and integration with HTML and CSS make it essential for front-end and increasingly for full-stack development.

Introduction

JavaScript is the only programming language that runs both in the browser and on the server, thanks to Node.js. It's so versatile that it can be used for web, mobile, and desktop apps, and even control hardware like robots and IoT devices. Its flexibility makes it a powerful tool for full-stack development and beyond!

Contents

1 Variables and Data Types

1.1 Variables in JavaScript

Variables are used to store data that can be referenced and manipulated in a program.

Declaring Variables

You can declare variables using three keywords:

- `var` (legacy, less commonly used now)
- `let` (preferred for mutable variables)
- `const` (for constant variables)

```
// Example
let x = 10;   // A variable that can be changed
const y = 20; // A constant variable
```

- **var**: Function scoped. Can lead to unexpected behavior.
- **let**: Block scoped. Safer and more predictable.
- **const**: Block scoped. Used for constants (though object contents can still change).

1.2 Data Types in JavaScript

JavaScript has a variety of built-in data types categorized as **primitive** and **non-primitive** types.

Primitive Data Types

These are immutable types.

1. **String**: Represents text.
   - Example: `"Hello"`, `'World'`, `` `Template` ``
2. **Number**: Represents integer and floating-point numbers.
   - Example: `42`, `3.14`
3. **BigInt**: Represents large integers.
   - Example: `12345678901234567890n`
4. **Boolean**: Represents logical values.
   - Example: `true`, `false`
5. **Undefined**: A variable declared but not assigned.
   - Example: `let x; console.log(x); // undefined`
6. **Null**: Represents no value.
   - Example: `let x = null;`
7. **Symbol**: Unique, immutable value.
   - Example: `let sym = Symbol("desc");`

Non-Primitive Data Types

1. **Object**: Key-value pair collections.

```
let person = { name: "John", age: 30 };
```

2. **Array**: Ordered collections.

```
let fruits = ["apple", "banana", "cherry"];
```

3. **Function**: Reusable blocks of code.

```
function greet(name) {
  return "Hello, " + name;
}
```

1.3 Type Conversion

JavaScript allows dynamic typing and automatic type conversion.

Implicit Type Conversion

```
let result = "5" + 3;  // "53"
let sum = "5" - 3;     // 2
```

Explicit Type Conversion

```
let num = "123";
let convertedNum = Number(num); // 123
```

1.4 Type Checking

You can check types using:

typeof

```
console.log(typeof 42);        // "number"
console.log(typeof "Hello");   // "string"
console.log(typeof true);      // "boolean"
console.log(typeof undefined); // "undefined"
console.log(typeof null);      // "object" (JavaScript quirk)
```

instanceof

```
let date = new Date();
console.log(date instanceof Date); // true
```

1.5 Example: Using Variables and Data Types

```
let age = 25;
let name = "Alice";
let isStudent = true;
let salary = null;
let undefinedVar;
let largeNumber = 9007199254740991n;

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

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

console.log(typeof age);       // "number"
console.log(typeof name);      // "string"
console.log(typeof isStudent); // "boolean"
console.log(typeof salary);    // "object"
console.log(typeof undefinedVar); // "undefined"
console.log(typeof largeNumber); // "bigint"
console.log(typeof person);    // "object"
console.log(typeof fruits);    // "object"
```

1.6 Conclusion

Understanding variables and data types in JavaScript is crucial for writing clean and effective code. By mastering these fundamentals, you can create dynamic and robust applications.

2 Operators in JavaScript

In JavaScript, operators are symbols that perform operations on variables and values. JavaScript supports a wide range of operators for performing arithmetic, logical, comparison, bitwise, and other operations. Here's an overview of the most common categories of operators in JavaScript:

2.1 Arithmetic Operators

These operators are used to perform basic arithmetic operations:

Addition

The `+` operator adds two operands.

Subtraction

The `-` operator subtracts the second operand from the first.

Multiplication

The `*` operator multiplies two operands.

Division

The `/` operator divides the first operand by the second.

Modulus

The `%` operator returns the remainder of a division.

Increment

The `++` operator increases the value of a variable by 1.

Decrement

The `--` operator decreases the value of a variable by 1.

2.2 Assignment Operators

These operators are used to assign values to variables:

Equal to Assignment

The `=` operator assigns the right-hand operand to the left-hand variable.

Add and Assign

The `+=` operator adds the right-hand operand to the left-hand operand and assigns the result to the left-hand operand.

Subtract and Assign

The `-=` operator subtracts the right-hand operand from the left-hand operand and assigns the result to the left-hand operand.

Multiply and Assign

The `*=` operator multiplies the left-hand operand by the right-hand operand and assigns the result to the left-hand operand.

Divide and Assign

The `/=` operator divides the left-hand operand by the right-hand operand and assigns the result to the left-hand operand.

Modulus and Assign

The `%=` operator assigns the remainder of the division of the left-hand operand by the right-hand operand to the left-hand operand.

2.3 Comparison Operators

These operators are used to compare two values and return a boolean result (`true` or `false`):

Equal to

The `==` operator compares values, ignoring data types.

Strict Equal to

The `===` operator compares both value and type.

Not Equal to

The `!=` operator compares values, ignoring data types.

Strict Not Equal to

The `!==` operator compares both value and type.

Greater Than

The `>` operator returns `true` if the left-hand operand is greater than the right-hand operand.

Less Than

The `<` operator returns `true` if the left-hand operand is less than the right-hand operand.

Greater Than or Equal to

The `>=` operator returns `true` if the left-hand operand is greater than or equal to the right-hand operand.

Less Than or Equal to

The `<=` operator returns `true` if the left-hand operand is less than or equal to the right-hand operand.

2.4 Logical Operators

These operators are used to perform logical operations, primarily with boolean values:

Logical AND

The operator returns `true` if both operands are `true`.

Logical OR

The `||` operator returns `true` if at least one operand is `true`.

Logical NOT

The `!` operator reverses the boolean value (true becomes false and vice versa).

2.5 Bitwise Operators

These operators perform bit-level operations on integer values:

Bitwise AND

The `&` operator performs a bitwise AND operation on two operands.

Bitwise OR

The `|` operator performs a bitwise OR operation on two operands.

Bitwise XOR

The $operatorperformsabitwiseXORoperationontwooperands.$

Bitwise NOT

`The ~ operator inverts the bits of a value.`

Left Shift

`The << operator shifts bits to the left by the specified number of positions.`

Right Shift

`The >> operator shifts bits to the right by the specified number of positions.`

Unsigned Right Shift

`The >>> operator shifts bits to the right, filling the leftmost bits with zeros.`

2.6 String Operators

Concatenation

`The + operator is used to combine (concatenate) two strings.`

Concatenation Assignment

`The += operator adds a string to an existing string.`

2.7 Ternary Operator

`The ternary operator is a shorthand for an if-else statement:`

Syntax

```
condition ?  expr1 :  expr2
If the condition is true, expr1 is executed; otherwise, expr2 is executed.
```

2.8 Type Operators

These operators are used to check or convert types:

Typeof

The typeof operator returns the type of a variable or expression.

Instanceof

The instanceof operator tests whether an object is an instance of a particular class or constructor function.

2.9 Unary Operators

These operators work on a single operand:

Unary Plus

The + operator converts its operand to a number.

Unary Minus

The - operator converts its operand to a number and negates it.

Increment

The ++ operator increases the operand by 1.

Decrement

The -- operator decreases the operand by 1.

Logical NOT

The ! operator reverses the boolean value of the operand.

2.10 Spread and Rest Operators

These operators are used in different contexts to manipulate arrays or objects:

Spread Operator

The ... (Spread) operator expands elements of an array or object.

Rest Operator

The ... (Rest) operator collects multiple values into an array in function arguments.

2.11 Nullish Coalescing Operator

The ?? operator returns the right-hand operand if the left-hand operand is null or undefined, otherwise it returns the left-hand operand.

2.12 Optional Chaining Operator

The ?. operator allows accessing deeply nested properties of an object without having to check each level for existence. If a reference is null or undefined, it returns undefined instead of throwing an error.

2.13 Example of Usage

```
let a = 10;
let b = 5;
let c = "Hello";
let result = a + b;  // 15
let str = c + " World";  // "Hello World"

console.log(result);  // Output: 15
console.log(str);  // Output: "Hello World"
```

These are the basic and most commonly used operators in JavaScript, which provide a wide variety of functionalities for manipulating data in programs.

3 Control structure in Javascript

In JavaScript, control structures are used to control the flow of the program. They help direct how and when certain code is executed based on conditions, loops, or functions. There are several types of control structures in JavaScript:

3.1 Conditional Statements

Conditional statements allow you to execute a block of code based on whether a condition is true or false.

3.1.1 'if' Statement

The if statement is used to execute a block of code if the condition is true.

```
let x = 10;
if (x > 5) {
  console.log("x is greater than 5");
}
```

3.1.2 'if...else' Statement

The if...else statement executes one block of code if the condition is true, and another block if the condition is false.

```
let x = 4;
if (x > 5) {
  console.log("x is greater than 5");
} else {
  console.log("x is not greater than 5");
}
```

3.1.3 'else if' Statement

The else if statement is used to specify a new condition if the previous if or else if conditions are false.

```
let x = 7;
if (x > 10) {
  console.log("x is greater than 10");
} else if (x === 7) {
  console.log("x is 7");
} else {
  console.log("x is less than 7");
}
```

3.1.4 'switch' Statement

The switch statement is used to execute one of many blocks of code based on the value of a variable or expression.

```
let day = 2;
switch (day) {
  case 1:
    console.log("Monday");
    break;
```

```
  case 2:
    console.log("Tuesday");
    break;
  case 3:
    console.log("Wednesday");
    break;
  default:
    console.log("Invalid day");
}
```

3.2 Looping Statements

Loops allow you to execute a block of code multiple times.

3.2.1 'for' Loop

The for loop is used when the number of iterations is known in advance.

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
// Output: 0, 1, 2, 3, 4
```

3.2.2 'while' Loop

The while loop repeats as long as the condition is true. It's used when the number of iterations is not known ahead of time.

```
let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
// Output: 0, 1, 2, 3, 4
```

3.2.3 'do...while' Loop

The do...while loop executes the code block once before checking the condition. Then, it repeats the loop as long as the condition is true.

```
let i = 0;
do {
  console.log(i);
  i++;
} while (i < 5);
// Output: 0, 1, 2, 3, 4
```

3.3 Jump Statements

Jump statements are used to control the flow of a program in a non-sequential way.

3.3.1 'break' Statement

The break statement is used to exit a loop or a switch statement.

```
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    break;
  }
  console.log(i);
}
// Output: 0, 1, 2, 3, 4
```

3.3.2 'continue' Statement

The continue statement is used to skip the current iteration of a loop and continue with the next iteration.

```
for (let i = 0; i < 5; i++) {
  if (i === 3) {
    continue; // skip the iteration when i is 3
  }
  console.log(i);
}
// Output: 0, 1, 2, 4
```

3.4 Exception Handling

JavaScript provides a way to handle errors or exceptional situations using try...catch statements.

3.4.1 'try...catch' Statement

The try...catch block allows you to handle exceptions (errors) that may occur during the execution of a block of code.

```
try {
  let result = someUndefinedFunction();
} catch (error) {
  console.log("An error occurred:", error.message);
} finally {
  console.log("This block is always executed.");
}
```

try: The code that may throw an exception is placed inside the try block.

catch: If an error occurs in the try block, the catch block will be executed.

finally: This block will execute after try or catch, regardless of whether an error was thrown or not.

3.5 Ternary Operator

The ternary operator (? :) is a shorthand for an if...else statement. It evaluates a condition and returns one of two values depending on whether the condition is true or false.

```
let x = 10;
let result = (x > 5) ? "x is greater than 5" : "x is less than or equal to 5";
console.log(result);  // Output: x is greater than 5
```

3.6 Short-circuiting with and ||

JavaScript allows logical operators to short-circuit based on the value of operands.

3.6.1 (AND)

If the first operand is false, the second operand will not be evaluated.

```
let x = 0;
let result = x && "This won't be evaluated";  // result is 0 because x is falsy
```

3.6.2 || (OR)

If the first operand is true, the second operand will not be evaluated.

```
let x = 5;
let result = x || "This won't be evaluated";  // result is 5 because x is truthy
```

3.7 Conclusion

These control structures are essential in JavaScript for managing the flow of execution in programs. They enable decision-making, repetition of code, handling exceptions, and controlling program execution with greater flexibility.

4 Functions in Javascript

In JavaScript, functions are fundamental building blocks that allow you to organize your code, reuse logic, and perform tasks. Functions in JavaScript can be declared in various ways and serve as a powerful feature for modular programming. Here's an overview of how functions work in JavaScript:

4.1 Function Declaration

A function declaration defines a named function that can be called later. The syntax is:

```
function functionName(parameters) {
    // function body
    return value;  // optional
}
```

4.1.1 Example

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

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

4.2 Function Expression

A function can also be assigned to a variable, known as a function expression. These functions can be anonymous or named.

```
const multiply = function mul(a, b) {
    return a * b;
};

console.log(multiply(4, 5));  // Output: 20
console.log(mul(2, 5));  // Output: 10
```

4.2.1 Named Function Expression

```
const divide = function div(a, b) {
    return a / b;
};

console.log(divide(20, 5));  // Output: 4
console.log(div(10, 2));  // Output: 5
```

4.2.2 Anonymous Function Expression

```
const subtract = function(a, b) {
    return a - b;
};
console.log(substract(10, 5));  // Output: 5
```

4.3 Arrow Functions

Arrow functions provide a shorter syntax for writing function expressions.

```
const divide = (a, b) => a / b;

console.log(divide(10, 2));  // Output: 5
```

4.3.1 Single Parameter Example

```
const square = x => x * x;
console.log(square(4));  // Output: 16
```

4.3.2 No Parameter Example

```
const greet = () => 'Hello, World!';
console.log(greet());  // Output: 'Hello, World!'
```

4.4 Immediately Invoked Function Expressions (IIFE)

An IIFE is a function that runs as soon as it is defined.

```
(function() {
    console.log('IIFE executed!');
})();
```

4.5 Function Parameters and Arguments

Functions in JavaScript can accept parameters and arguments.

4.5.1 Example

```
function greet(name, age) {
    console.log(`Hello, my name is ${name} and I am ${age} years old.`);
}

greet('Alice', 30);  // Output: Hello, my name is Alice and I am 30 years old.
```

4.5.2 Default Parameters

```
function greet(name = 'Guest', age = 25) {
    console.log(`Hello, my name is ${name} and I am ${age} years old.`);
}

greet();  // Output: Hello, my name is Guest and I am 25 years old.
```

4.5.3 Rest Parameters

```
function sum(...numbers) {
    return numbers.reduce((acc, curr) => acc + curr, 0);
}

console.log(sum(1, 2, 3, 4));  // Output: 10
```

4.6 Return Statement

Functions can return values using the return keyword.

```
function multiply(a, b) {
    return a * b;
}

console.log(multiply(3, 4));  // Output: 12
```

4.7 Function Scope and Closures

Functions create their own scope, and closures allow them to remember variables from their outer scope.

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

    function inner() {
        console.log(outerVar);  // inner() can access outerVar
    }

    inner();
}

outer();  // Output: 'I am from outer scope'
```

4.8 Higher-Order Functions

A higher-order function takes one or more functions as arguments or returns a function.

```
function applyFunction(f, x) {
    return f(x);
}

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

console.log(applyFunction(double, 5));  // Output: 10
```

4.9 Anonymous Functions

Anonymous functions are defined without a name and are often used as callbacks.

```
setTimeout(function() {
    console.log('This is an anonymous function');
}, 1000);
```

4.10 Recursive Functions

A function can call itself to perform recursion.

```
function factorial(n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}

console.log(factorial(5));  // Output: 120
```

4.11 Summary of Key Points

4.11.1 Function Declaration

Named function that can be called anywhere in the code.

4.11.2 Function Expression

A function assigned to a variable; can be anonymous.

4.11.3 Arrow Functions

Shorter syntax for function expressions.

4.11.4 IIFE

A function that runs immediately after being defined.

4.11.5 Parameters and Arguments

You can pass values to functions; default and rest parameters provide flexibility.

4.11.6 Return Statement

Functions can return values, or return undefined by default.

4.11.7 Closures

Functions can remember and access variables from their outer scope.

4.11.8 Higher-Order Functions

Functions that take other functions as arguments or return functions.

4.11.9 Anonymous Functions

Functions without a name, often used as callbacks.

4.11.10 Recursion

Functions that call themselves.

5 Objects and arrays in JS

In JavaScript, objects and arrays are two fundamental data structures used to store collections of values. Here's a breakdown of how they work:

5.1 Objects

An object is a collection of key-value pairs. Each key (or property) is a string or symbol, and each value can be any JavaScript data type.

5.1.1 Syntax

```
const person = {
  name: "John",
  age: 30,
  isStudent: false,
  greet: function() {
    console.log("Hello!");
  }
};
```

Keys are strings or symbols, and values can be of any type including strings, numbers, arrays, functions, or objects.

5.1.2 Accessing Object Properties

```
// Dot notation
console.log(person.name);  // Output: John

// Bracket notation
console.log(person["age"]);  // Output: 30
```

Use dot notation for standard identifiers and bracket notation for dynamic or invalid identifiers.

5.1.3 Adding or Modifying Properties

```
// Adding a new property
person.city = "New York";

// Modifying an existing property
person.age = 31;
```

5.1.4 Deleting Properties

```
delete person.isStudent;
```

5.1.5 Example: Object with Mixed Data Types

```
const car = {
  make: "Toyota",
  model: "Corolla",
  year: 2022,
  features: ["Bluetooth", "Backup Camera", "Sunroof"],
  start: function() {
    console.log("Engine started");
  }
};
```

5.2 Arrays

An array is a special object that holds ordered collections of values indexed numerically starting from 0.

5.2.1 Syntax

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

Arrays can contain any type of data including objects, arrays, or functions.

5.2.2 Accessing Array Elements

```
console.log(fruits[0]);  // Output: apple
console.log(fruits[2]);  // Output: cherry
```

5.2.3 Modifying Array Elements

```
fruits[1] = "blueberry";
```

5.2.4 Array Methods

```
fruits.push("orange");
```

push() – Add to end

```
fruits.pop();
```

pop() – Remove from end

```
fruits.shift();
```

shift() – Remove from start

```
fruits.unshift("kiwi");
```

unshift() – Add to start

```
fruits.forEach(function(fruit) {
  console.log(fruit);
});
```

forEach() – Iterate

```
const uppercaseFruits = fruits.map(fruit => fruit.toUpperCase());
```

map() – Transform

```
const longFruits = fruits.filter(fruit => fruit.length > 5);
```

filter() – Filter by condition

```
const foundFruit = fruits.find(fruit => fruit === "banana");
```

find() – Find first match

5.2.5 Example: Array with Mixed Data Types

```
const mixedArray = [42, "hello", true, { name: "Alice" }, [1, 2, 3]];
```

5.3 Key Differences Between Objects and Arrays

Feature	**Objects**	**Arrays**
Purpose	Used to store data as key-value pairs.	Used to store ordered lists of items.
Indexing	Keys can be strings (or symbols).	Indexed numerically (0, 1, 2, ...).
Accessing values	`object.key` or `object["key"]`	`array[index]`
Order	No guaranteed order for keys.	Ordered, elements are indexed numerically.
Best Use Case	Representing entities (e.g., a person, a car).	Representing lists (e.g., a list of numbers, fruits).

Table 1: Comparison of Objects and Arrays

5.4 Summary

- `Objects are ideal for grouping related data with named keys.`
- `Arrays are perfect for ordered collections of items.`
- `Use objects when the data has named attributes.`
- `Use arrays when the data has a natural order.`

6 Events and Event Handling in JavaScript

In JavaScript, events are actions or occurrences that take place in the browser, such as clicking a button, submitting a form, or resizing the window. Developers can write code to respond to these events, enabling dynamic, interactive web pages.

6.1 Common Types of Events

Here are some common categories of events:

- Mouse Events: click, dblclick, mousemove, mouseover, mouseout, mousedown, mouseup
- Keyboard Events: keydown, keypress, keyup
- Form Events: submit, change, focus, blur, input
- Window/Document Events: load, resize, scroll, DOMContentLoaded
- Touch Events (Mobile): touchstart, touchmove, touchend, touchcancel

6.2 Ways to Handle Events

6.2.1 1. Inline Event Handlers (HTML)

```
<button onclick="alert('Button clicked!')">Click Me</button>
```

This method mixes HTML with JavaScript and is not recommended for maintainable code.

6.2.2 2. addEventListener (JavaScript)

```
<button id="myButton">Click Me</button>

<script>
  const button = document.getElementById('myButton');
  button.addEventListener('click', function() {
    alert('Button clicked!');
  });
</script>
```

This is the preferred method as it allows multiple event listeners and cleaner separation of code.

6.2.3 3. Event Handler Properties

```
<button id="myButton">Click Me</button>

<script>
  const button = document.getElementById('myButton');
  button.onclick = function() {
    alert('Button clicked!');
  };
</script>
```

This method is simple but allows only one handler per event type on an element.

6.3 Event Propagation

6.3.1 1. Event Bubbling (Default)

```
document.getElementById('myButton').addEventListener('click', function(event) {
  console.log('Button clicked!');
  event.stopPropagation();  // Stops bubbling
});

document.getElementById('myDiv').addEventListener('click', function() {
  console.log('Div clicked!');
});
```

6.3.2 2. Event Capturing

```
document.getElementById('myDiv').addEventListener('click', function() {
  console.log('Div clicked!');
}, true);  // Capturing mode
```

6.4 The Event Object

When an event is triggered, an event object is passed to the handler:

```
document.getElementById('myButton').addEventListener('click', function(event) {
  console.log('Event type:', event.type);
  console.log('Event target:', event.target);
});
```

Common Properties:

- event.type: Type of the event (e.g., click)
- event.target: The element that triggered the event
- event.preventDefault(): Prevents default behavior (e.g., form submission)
- event.stopPropagation(): Stops the event from bubbling
- event.keyCode / event.code: Keyboard input details
- event.clientX / event.clientY: Mouse position

6.5 Removing Event Listeners

To remove an event listener, you must reference the same function:

```
const button = document.getElementById('myButton');

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

button.addEventListener('click', onClickHandler);

// Later...
button.removeEventListener('click', onClickHandler);
```

6.6 Event Delegation

Instead of adding listeners to each child, add one to a parent and use event.target:

```
<ul id="myList">
  <li>Item 1</li>
  <li>Item 2</li>
</ul>

<script>
  const list = document.getElementById('myList');

  list.addEventListener('click', function(event) {
    if (event.target.tagName === 'LI') {
      alert('Clicked: ' + event.target.textContent);
    }
  });
</script>
```

This is useful for dynamic content where items are added or removed.

6.7 Conclusion

JavaScript events are essential for building interactive web applications. By mastering event handling techniques such as addEventListener, understanding propagation, and using delegation, developers can create responsive and maintainable code.

7 Asynchronous programming in JavaScript

Asynchronous programming in JavaScript allows you to write code that performs tasks without blocking the main execution thread. This is particularly important in scenarios where you need to handle I/O operations like reading files, making HTTP requests, or interacting with databases, without freezing the user interface or slowing down the application.

JavaScript provides several mechanisms to handle asynchronous code execution. Let's dive into the most commonly used techniques:

7.1 Callbacks

A callback is a function that is passed as an argument to another function, which is executed once an operation completes.

7.1.1 Example:

```
function fetchData(callback) {
    setTimeout(() => {
        console.log("Data fetched");
        callback();
    }, 1000);
}

fetchData(() => {
    console.log("Callback executed after fetching data.");
});
```

In this example, fetchData accepts a callback that is executed after the asynchronous operation (setTimeout) completes. While simple, this approach can lead to "callback hell" in more complex scenarios.

7.2 Promises

A Promise is an object representing the eventual completion (or failure) of an asynchronous operation. Promises have three possible states:

- Pending: The operation is still in progress.
- Fulfilled: The operation completed successfully.
- Rejected: The operation failed.

7.2.1 Example:

```
function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const success = true;
            if (success) {
                resolve("Data fetched");
            } else {
                reject("Error fetching data");
            }
        }, 1000);
    });
}

fetchData()
```

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

7.3 Async/Await

The async and await keywords provide a more readable, synchronous-looking syntax for working with promises.

- The async keyword is used to define a function that returns a promise.
- The await keyword pauses execution until the promise resolves.

7.3.1 Example:

```
function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const success = true;
            if (success) {
                resolve("Data fetched");
            } else {
                reject("Error fetching data");
            }
        }, 1000);
    });
}

async function getData() {
    try {
        const data = await fetchData();
        console.log(data);
    } catch (error) {
        console.error(error);
    }
}

getData();
```

7.4 Chaining Promises

Promises can be chained to perform multiple asynchronous operations in sequence.

7.4.1 Example:

```
function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => resolve("Data fetched"), 1000);
    });
}

function processData(data) {
    return new Promise((resolve) => {
        setTimeout(() => resolve(data.toUpperCase()), 1000);
    });
}

fetchData()
    .then(data => processData(data))
```

```
    .then(processedData => console.log(processedData))
    .catch(error => console.error(error));
```

7.5 Error Handling

Proper error handling is essential:

- With callbacks: pass the error as the first argument.
- With promises: use .catch().
- With async/await: use try/catch.

7.6 Concurrency and Parallelism

You can run multiple asynchronous operations in parallel using utility functions like Promise.all and Promise.race.

7.6.1 Example: Promise.all

```
const fetchData1 = () => new Promise(resolve => setTimeout(() => resolve("Data 1"), 1000));
const fetchData2 = () => new Promise(resolve => setTimeout(() => resolve("Data 2"), 500));

Promise.all([fetchData1(), fetchData2()])
    .then(([data1, data2]) => {
        console.log(data1);
        console.log(data2);
    })
    .catch(error => console.error(error));
```

7.6.2 Example: Promise.race

```
Promise.race([fetchData1(), fetchData2()])
    .then(result => console.log(result))
    .catch(error => console.error(error));
```

7.7 setTimeout and setInterval with Promises

You can wrap setTimeout in a promise to delay execution:

7.7.1 Example:

```
function delay(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}

async function run() {
    console.log("Start");
    await delay(2000);
    console.log("After 2 seconds");
}

run();
```

7.8 Conclusion

- Callbacks: Simple, but can lead to deeply nested code.
- Promises: Cleaner syntax for managing async operations.
- Async/Await: Most readable and intuitive for asynchronous logic.

By understanding and using these tools effectively, you can write more efficient and maintainable asynchronous JavaScript code.

8 Scope and Closures in Javascript

In JavaScript, scope and closures are two key concepts that play a crucial role in how variables and functions interact. Understanding them is essential for writing efficient, bug-free code.

8.1 Scope in JavaScript

Scope refers to the context in which variables are declared and where they are accessible. JavaScript has several types of scopes:

8.1.1 Global Scope

- Variables declared outside of any function or block are said to be in the global scope.
- Global variables are accessible from anywhere in the code, even inside functions.

Example:

```
var globalVar = "I am global";

function example() {
  console.log(globalVar); // Accessible
}

example(); // Output: "I am global"
```

8.1.2 Function Scope

- Variables declared inside a function are within the function scope. They are only accessible inside that function.
- A function scope is created when a function is invoked, and variables declared inside it are not accessible outside.

Example:

```
function myFunction() {
  var functionVar = "I am local";
  console.log(functionVar); // Accessible inside the function
}

console.log(functionVar); // Error: functionVar is not defined outside the function
```

8.1.3 Block Scope (introduced in ES6)

- Variables declared with let or const are scoped to the block they are declared in, which can be a loop, conditional statement, or any curly-braced block.
- Unlike var, which is function-scoped, let and const are block-scoped.

```
if (true) {
  let blockVar = "I am block-scoped";
  console.log(blockVar); // Accessible inside the block
}

console.log(blockVar); // Error: blockVar is not defined outside the block
```

Example:

8.1.4 Lexical Scope

- JavaScript uses lexical scoping, meaning that the scope of a variable is determined by where it is defined in the source code, not where it is called.

```
function outer() {
  var outerVar = "I am from the outer function";

  function inner() {
    console.log(outerVar); // inner can access outer's scope
  }

  inner(); // Output: "I am from the outer function"
}

outer();
```

Example:

8.2 Closures in JavaScript

A closure is created when a function retains access to its lexical scope, even after the function has finished executing. This means that inner functions can access variables from their outer (enclosing) functions, even after the outer function has returned.

Closures are fundamental to JavaScript, and they enable powerful patterns like data privacy, currying, and maintaining state in asynchronous code.

8.2.1 Example of a Closure

```
function outerFunction() {
  var outerVar = "I am from the outer function";

  return function innerFunction() {
    console.log(outerVar); // innerFunction can access outerVar
  };
}

var closureExample = outerFunction();
closureExample(); // Output: "I am from the outer function"
```

In this example, innerFunction is a closure that "remembers" the scope of outerFunction, even though outerFunction has finished executing. The variable outerVar is still accessible to innerFunction.

8.2.2 Closures and Private Data

Closures allow JavaScript to create private variables. A common pattern is using closures to encapsulate data and control access to it via getter/setter functions.

```
function createCounter() {
  let count = 0; // count is private and not directly accessible outside

  return {
    increment: function() {
      count++;
      console.log(count);
    },
    decrement: function() {
      count--;
      console.log(count);
    },
    getCount: function() {
      return count;
    }
  };
}

const counter = createCounter();
counter.increment(); // Output: 1
counter.increment(); // Output: 2
console.log(counter.getCount()); // Output: 2
counter.decrement(); // Output: 1
```

Example:
Here, the count variable is enclosed within the createCounter function and cannot be accessed directly from outside. The functions increment, decrement, and getCount form closures, giving them access to count.

8.2.3 Closures in Asynchronous Code

Closures are often used in asynchronous code, like in event handlers or setTimeout, where the inner function needs to access variables that were defined when the asynchronous task was set up.

```
function delayMessage(message, delay) {
  setTimeout(function() {
    console.log(message); // Closure captures `message` and `delay`
  }, delay);
}

delayMessage("Hello after 2 seconds", 2000);
```

Example:
In this case, the inner function (the callback passed to setTimeout) forms a closure that "remembers" the message and delay values, even after delayMessage has finished executing.

8.3 Summary of Scope and Closures

- Scope defines where a variable is accessible in your code (global, function, block).
- Lexical Scope refers to how a function's scope is determined by where it is defined.
- A closure is created when an inner function retains access to variables from its outer (enclosing) function, even after the outer function has returned.

- Closures are useful for data encapsulation, creating private variables, and maintaining state, especially in asynchronous code.

Mastering scope and closures will help you write more efficient, modular, and maintainable JavaScript code.

9 Error Handling

Error handling in JavaScript is a crucial part of writing reliable and robust applications. JavaScript provides mechanisms to deal with errors that occur during the execution of the program. The primary tools for error handling are:

9.1 Try-Catch Statement

The try-catch statement is the most common way of handling errors in JavaScript. It allows you to "try" a block of code and "catch" any exceptions that occur.

9.1.1 Syntax:

```
try {
  // Code that may throw an error
} catch (error) {
  // Code that runs if an error occurs
  console.error(error); // Handling the error
} finally {
  // Optional block that runs after try and catch, regardless of success or failure
  console.log("This always runs.");
}
```

- try: Contains code that might throw an error.
- catch: Executes if an error occurs in the try block. The error is captured as an object (commonly called error or e).
- finally: This block is optional and runs regardless of whether an error occurred or not, making it useful for clean-up actions (like closing file handlers or database connections).

9.1.2 Example:

```
try {
  let result = riskyFunction();
  console.log(result);
} catch (error) {
  console.error("An error occurred: ", error.message);
} finally {
  console.log("Cleanup or final steps.");
}
```

9.2 Throwing Errors

You can throw custom errors using the throw keyword. This is useful when you want to raise an error intentionally if something goes wrong in your logic or if input validation fails.

9.2.1 Syntax:

```
throw new Error('Something went wrong');
```

9.2.2 Example:

```
function checkAge(age) {
  if (age < 18) {
    throw new Error("Age must be 18 or older.");
  }
  return "Age is valid";
}

try {
  console.log(checkAge(15));
} catch (error) {
  console.error(error.message); // "Age must be 18 or older."
}
```

9.3 Error Object

When an error occurs in JavaScript, an Error object is generated. You can use this object to get more information about the error, such as its message, stack trace, and name.

- name: The name of the error (e.g., Error, TypeError, ReferenceError).
- message: A human-readable description of the error.
- stack: A stack trace that shows where the error occurred.

9.3.1 Example:

```
try {
  throw new TypeError("This is a type error");
} catch (error) {
  console.log(error.name);    // TypeError
  console.log(error.message); // This is a type error
  console.log(error.stack);   // Stack trace (for debugging)
}
```

9.4 Asynchronous Error Handling (Promises & Async/Await)

In modern JavaScript, errors can also occur in asynchronous code. These are typically handled using Promises and async/await syntax.

9.4.1 Promises:

With promises, you can handle errors using the .catch() method.

```
function fetchData() {
  return new Promise((resolve, reject) => {
    let success = false;
    if (success) {
      resolve("Data fetched successfully");
    } else {
      reject("Failed to fetch data");
    }
  });
}

fetchData()
  .then(result => console.log(result))
  .catch(error => console.error(error));
```

9.4.2 Async/Await:

Using async functions with await, you can use try-catch for handling errors in asynchronous code.

```
async function fetchData() {
  let success = false;
  if (success) {
    return "Data fetched successfully";
  } else {
    throw new Error("Failed to fetch data");
  }
}

async function getData() {
  try {
    const result = await fetchData();
    console.log(result);
  } catch (error) {
    console.error(error.message); // Handle the error
  }
}

getData();
```

9.5 Custom Error Types

You can create custom error types by extending the Error class. This is useful when you need more specific error handling, such as distinguishing between different kinds of errors in your application.

9.5.1 Example:

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError"; // Custom error name
  }
}

function validateEmail(email) {
  if (!email.includes('@')) {
    throw new ValidationError("Invalid email address");
  }
  return "Email is valid";
}

try {
  console.log(validateEmail("invalidEmail"));
} catch (error) {
  if (error instanceof ValidationError) {
    console.error(`Validation failed: ${error.message}`);
  } else {
    console.error(`General error: ${error.message}`);
  }
}
```

9.6 Handling Multiple Errors (Aggregate Errors)

In some cases, you might want to handle multiple errors at once, for example, when working with Promise.all and catching multiple rejections.

```
const promise1 = Promise.reject(new Error("Promise 1 failed"));
const promise2 = Promise.reject(new Error("Promise 2 failed"));

Promise.allSettled([promise1, promise2])
  .then(results => {
    results.forEach(result => {
      if (result.status === "rejected") {
        console.error(result.reason.message);
      }
    });
  });
```

9.7 Global Error Handling

JavaScript also provides global error handling mechanisms for unhandled errors.

- window.onerror (Browser): Catches global errors in the browser.

```
window.onerror = function(message, source, lineno, colno, error) {
  console.error(`Error: ${message} at ${source}:${lineno}:${colno}`);
  return true; // Prevents default handling (e.g., showing error in console)
};
```

- process.on('uncaughtException') (Node.js): Handles uncaught exceptions in Node.js.

```
process.on('uncaughtException', (error) => {
  console.error('Uncaught exception: ', error);
  process.exit(1); // Optionally exit the process
});
```

9.8 Best Practices for Error Handling:

- Graceful Error Reporting: Instead of crashing the application, catch and handle errors gracefully, logging them for debugging and providing useful feedback to users.
- Avoid Silent Failures: Don't ignore errors; always log them or handle them appropriately.
- Use Specific Errors: Instead of using a generic Error, create specific error types (e.g., ValidationError, DatabaseError) to make debugging easier.
- Use finally for Cleanup: If you have resource cleanup tasks, put them in the finally block to ensure they run regardless of success or failure.
- Test for Edge Cases: Ensure your error handling works even when unexpected or edge cases arise.

By following these error-handling techniques, you can build more resilient and maintainable JavaScript applications.

10 What is DOM Manipulation?

The DOM (Document Object Model) represents a web page as a tree of nodes. JavaScript can interact with and modify the structure, content, and style of the page dynamically.

10.1 Selecting Elements

Before manipulating elements, you must select them. JavaScript provides several methods:

- getElementById(id)

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

- getElementsByClassName(className)

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

- getElementsByTagName(tagName)

```
const paragraphs = document.getElementsByTagName('p');
```

- querySelector(selector)

```
const firstDiv = document.querySelector('div');
```

- querySelectorAll(selector)

```
const allLinks = document.querySelectorAll('a');
```

10.2 Modifying Elements

10.2.1 Changing Content

- innerHTML

```
element.innerHTML = 'New Content';
```

- textContent

```
paragraph.textContent = 'Updated text content';
```

10.2.2 Changing Attributes

- getAttribute()

```
const srcValue = img.getAttribute('src');
```

- setAttribute()

```
img.setAttribute('alt', 'New Alt Text');
```

- `removeAttribute()`

```
img.removeAttribute('width');
```

10.2.3 Changing Styles

```
div.style.backgroundColor = 'blue';
div.style.fontSize = '20px';
```

10.2.4 Modifying Classes

```
element.classList.add('new-class');
element.classList.remove('old-class');
element.classList.toggle('highlight');

if (element.classList.contains('highlight')) {
  console.log('Element is highlighted');
}
```

10.3 Creating New Elements

```
const newDiv = document.createElement('div');
newDiv.textContent = 'This is a new div element';

const container = document.getElementById('container');
container.appendChild(newDiv);

const referenceNode = document.querySelector('.reference');
container.insertBefore(newDiv, referenceNode);
```

10.4 Removing Elements

```
const parent = document.getElementById('parent');
const child = document.getElementById('child');
parent.removeChild(child);

const element = document.getElementById('element');
element.remove();
```

10.5 Event Handling

```
const button = document.getElementById('myButton');
button.addEventListener('click', function() {
  alert('Button clicked!');
});

button.removeEventListener('click', myFunction);
```

10.6 Traversing the DOM

```
const parent = child.parentNode;
const children = parent.childNodes;

const first = parent.firstChild;
const last = parent.lastChild;

const next = child.nextSibling;
const prev = child.previousSibling;
```

10.7 Animating the DOM

- Use CSS by changing styles or toggling classes.
- Use requestAnimationFrame() for smooth JavaScript animations.

10.8 Example: DOM Manipulation

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>DOM Manipulation Example</title>
  <style>
    .highlight { color: red; }
  </style>
</head>
<body>

<h1 id="header">Hello, World!</h1>
<button id="changeTextBtn">Change Text</button>
<button id="toggleClassBtn">Toggle Highlight</button>
<div id="container">
  <p>This is a paragraph.</p>
</div>

<script>
  const header = document.getElementById('header');
  const changeTextBtn = document.getElementById('changeTextBtn');
  const toggleClassBtn = document.getElementById('toggleClassBtn');
  const container = document.getElementById('container');

  changeTextBtn.addEventListener('click', () => {
    header.textContent = 'Text Changed!';
  });

  toggleClassBtn.addEventListener('click', () => {
    header.classList.toggle('highlight');
  });

  const newParagraph = document.createElement('p');
  newParagraph.textContent = 'This is a dynamically added paragraph.';
  container.appendChild(newParagraph);
</script>

</body>
</html>
```

10.9 Summary

- Use DOM selectors to access elements.
- Modify content, attributes, styles, and classes.
- Use event listeners for interactivity.
- Create, remove, and animate elements dynamically.

11 ES6+ Features

DOM (Document Object Model) manipulation refers to the process of using JavaScript to interact with and modify the structure, content, and style of a web page dynamically. The DOM represents the page as a tree of nodes, where each node corresponds to part of the document (e.g., elements, attributes, text).

Here are the key operations and methods in JavaScript for DOM manipulation:

11.1 Selecting Elements

Before you can manipulate any element, you need to select it. There are several ways to select DOM elements in JavaScript:

getElementById(id) Selects an element by its ID.

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

getElementsByClassName(className) Selects all elements with the specified class name.

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

getElementsByTagName(tagName) Selects all elements with the specified tag name.

```
const paragraphs = document.getElementsByTagName('p');
```

querySelector(selector) Selects the first element that matches the given CSS selector.

```
const firstDiv = document.querySelector('div');
```

querySelectorAll(selector) Selects all elements that match the given CSS selector.

```
const allLinks = document.querySelectorAll('a');
```

11.2 Modifying Elements

Once you have selected an element, you can manipulate its content, attributes, and styles.

11.2.1 Changing Content

```
const element = document.getElementById('myElement');
element.innerHTML = 'New Content';
```

innerHTML

```
const paragraph = document.querySelector('p');
paragraph.textContent = 'Updated text content';
```

textContent

11.2.2 Changing Attributes

```
const img = document.querySelector('img');
const srcValue = img.getAttribute('src');
```

getAttribute(attribute)

```
img.setAttribute('alt', 'New Alt Text');
```

setAttribute(attribute, value)

```
img.removeAttribute('width');
```

removeAttribute(attribute)

11.2.3 Changing Styles

```
const div = document.querySelector('div');
div.style.backgroundColor = 'blue';
div.style.fontSize = '20px';
```

11.2.4 Modifying Classes

```
const element = document.querySelector('p');
element.classList.add('new-class');
```

classList.add(className)

```
element.classList.remove('old-class');
```

classList.remove(className)

```
element.classList.toggle('highlight');
```

classList.toggle(className)

```
if (element.classList.contains('highlight')) {
  console.log('Element is highlighted');
}
```

classList.contains(className)

11.3 Creating New Elements

```
const newDiv = document.createElement('div');
newDiv.textContent = 'This is a new div element';
```

createElement(tagName)

```
const container = document.getElementById('container');
container.appendChild(newDiv);
```

appendChild(child)

```
const referenceNode = document.querySelector('.reference');
container.insertBefore(newDiv, referenceNode);
```

insertBefore(newNode, referenceNode)

11.4 Removing Elements

```
const parent = document.getElementById('parent');
const child = document.getElementById('child');
parent.removeChild(child);
```

removeChild(child)

```
const element = document.getElementById('element');
element.remove();
```

remove()

11.5 Event Handling

```
const button = document.getElementById('myButton');
button.addEventListener('click', function() {
  alert('Button clicked!');
});
```

addEventListener(type, listener)

```
button.removeEventListener('click', myFunction);
```

removeEventListener(type, listener)

11.6 Traversing the DOM

```
const child = document.querySelector('div');
const parent = child.parentNode;
```

parentNode

```
const parent = document.querySelector('div');
const children = parent.childNodes;
```

childNodes

```
const first = parent.firstChild;
const last = parent.lastChild;
```

firstChild / lastChild

```
const next = child.nextSibling;
const prev = child.previousSibling;
```

nextSibling / previousSibling

11.7 Animating the DOM

You can also animate elements using CSS or JavaScript.

- CSS animations can be triggered by manipulating the style property, or by adding/removing CSS classes.
- JavaScript-based animations use requestAnimationFrame() for smooth frame-by-frame animations.

11.8 Example of DOM Manipulation

```
<!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; }
    </style>
</head>
<body>

<h1 id="header">Hello, World!</h1>
<button id="changeTextBtn">Change Text</button>
<button id="toggleClassBtn">Toggle Highlight</button>
<div id="container">
    <p>This is a paragraph.</p>
</div>

<script>
    const header = document.getElementById('header');
    const changeTextBtn = document.getElementById('changeTextBtn');
    const toggleClassBtn = document.getElementById('toggleClassBtn');
    const container = document.getElementById('container');

    changeTextBtn.addEventListener('click', () => {
        header.textContent = 'Text Changed!';
    });

    toggleClassBtn.addEventListener('click', () => {
        header.classList.toggle('highlight');
    });

    const newParagraph = document.createElement('p');
    newParagraph.textContent = 'This is a dynamically added paragraph.';
    container.appendChild(newParagraph);
</script>

</body>
</html>
```

11.9 Summary

- Selecting elements: Methods like getElementById, querySelector, etc.
- Modifying elements: Use innerHTML, textContent, setAttribute, etc.
- Event handling: Use addEventListener to handle user events.
- Creating and removing elements: Use createElement, appendChild, etc.

JavaScript allows you to make web pages interactive and responsive to user actions in real-time.

12 Class and Inheritance

In JavaScript, classes and inheritance are key concepts used to create and manage object-oriented designs. Classes provide a blueprint for creating objects, and inheritance allows one class to derive properties and methods from another, enabling code reuse and hierarchical relationships between objects.

12.1 Classes in JavaScript

JavaScript introduced the class syntax in ECMAScript 6 (ES6), but under the hood, it still uses prototypal inheritance. A class in JavaScript is a function that is used as a template to create objects with shared properties and methods.

12.1.1 Syntax for Defining a Class

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

  greet() {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
  }
}
```

In this example, the Person class has a constructor and a greet method available to all instances.

12.1.2 Creating Instances of a Class

```
const person1 = new Person('Alice', 30);
person1.greet();

const person2 = new Person('Bob', 25);
person2.greet();
```

12.2 Inheritance in JavaScript

Inheritance allows a class (child class) to inherit properties and methods from another class (parent class) using the extends keyword.

12.2.1 Syntax for Inheriting from a Parent Class

```
class Employee extends Person {
  constructor(name, age, jobTitle) {
    super(name, age);
    this.jobTitle = jobTitle;
  }

  describeJob() {
    console.log(`${this.name} works as a ${this.jobTitle}.`);
  }
}
```

The Employee class extends Person and includes a new method describeJob().

12.2.2 Creating an Instance of a Subclass

```
const employee1 = new Employee('Charlie', 40, 'Software Developer');
employee1.greet();
employee1.describeJob();
```

12.3 Method Overriding

A subclass can override a method from its parent class:

```
class Employee extends Person {
  constructor(name, age, jobTitle) {
    super(name, age);
    this.jobTitle = jobTitle;
  }

  greet() {
    console.log(`Hi, I'm ${this.name}, a ${this.jobTitle}, and I am ${this.age} years old.`);
  }
}

const employee2 = new Employee('David', 28, 'Graphic Designer');
employee2.greet();
```

12.4 Getter and Setter Methods

Getters and setters control access to object properties:

```
class Rectangle {
  constructor(width, height) {
    this._width = width;
    this._height = height;
  }

  get area() {
    return this._width * this._height;
  }

  set width(value) {
    if (value > 0) {
      this._width = value;
    }
  }

  set height(value) {
    if (value > 0) {
      this._height = value;
    }
  }
}

const rect = new Rectangle(5, 10);
console.log(rect.area);

rect.width = 8;
console.log(rect.area);
```

12.5 Static Methods

Static methods belong to the class, not the instances:

```
class MathUtility {
  static square(x) {
    return x * x;
  }

  static cube(x) {
    return x * x * x;
  }
}

console.log(MathUtility.square(4));
console.log(MathUtility.cube(3));
```

12.6 Private Fields (ES2022+)

Private fields, prefixed with #, can only be accessed within the class:

```
class Car {
  #speed;

  constructor(make, model) {
    this.make = make;
    this.model = model;
    this.#speed = 0;
  }

  accelerate() {
    this.#speed += 10;
    console.log(`The car is now going ${this.#speed} km/h.`);
  }

  brake() {
    this.#speed = 0;
    console.log('The car has stopped.');
  }
}

const myCar = new Car('Toyota', 'Corolla');
myCar.accelerate();
myCar.brake();

// console.log(myCar.#speed); // SyntaxError
```

12.7 Summary

- Classes define blueprints for objects.
- Inheritance allows reuse and extension of behavior.
- Method Overriding customizes parent methods.
- Getters and Setters provide controlled property access.
- Static Methods are called on the class, not instances.
- Private Fields enable true encapsulation in ES2022+.

JavaScript's class and inheritance system enables structured, reusable, and maintainable code in object-oriented programming.